ADI处理器实用丛书

数字视频处理原理及DSP实现

编著

电子工业出版社

fastbuk bookstore

ADI 处理器实用丛书

数字视频处理原理及 DSP 实现

邢延超　皇甫伟　编著

電子工業出版社
Publishing House of Electronics Industry
北京 • BEIJING

内 容 简 介

本书在介绍数字视频信号处理基本概念和常用算法的基础上，从实用性的角度出发，借助 ADI 公司的 Blackfin 系列 DSP 处理器平台，讨论了嵌入式视频处理的相关问题。主要内容包括：绪论、数字视频基础、数字信号处理与嵌入式开发、基于 Blackfin 处理器的最小视频系统、视频应用设计原则及基础应用简介、图像与视频处理软件开发包、视频运动分析及应用、视频编解码理论及实现、视频时空滤波及实现。后面几部分中包含了运动跟踪、H.264 编解码和视频去交错应用等具体应用。

本书可作为对视频处理和 DSP 开发感兴趣的科研技术人员的参考，也可作为信息与信号处理方向研究生和高年级本科生的参考书。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

数字视频处理原理及 DSP 实现 / 邢延超，皇甫伟编著. —北京：电子工业出版社，2011.12
（ADI 处理器实用丛书）
ISBN 978-7-121-15417-1

Ⅰ. ①数… Ⅱ. ①邢… ②皇… Ⅲ. ①数字视频系统—数字信号处理 Ⅳ. ①TN941.1

中国版本图书馆 CIP 数据核字（2011）第 253409 号

策划编辑：竺南直　　徐蔷薇
责任编辑：谭丽莎　　文字编辑：王凌燕
印　　刷：
装　　订：
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编 100036
开　　本：787×1 092　1/16　印张：18　字数：461 千字
印　　次：2011 年 12 月第 1 次印刷
印　　数：3 000 册　　定价：39.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

前　言

近年来，数字视频处理在智能监控、消费类电子、机器人视觉、工业检测与控制及网络视频会议等领域的应用发展迅猛。同时，许多视频应用产品都在朝着小型化、便携化、低功耗和易升级等方向发展。随着高速数字处理器技术的发展，嵌入式数字视频处理已经成为重要的发展方向，具有客观的市场前景。

DSP 不仅具有高性能的数字信号处理能力，同时还具有开发速度块、升级方便且成本低等明显优势，在该领域发挥着越来越重要的作用。Blackfin 系列 DSP 是 ADI 公司开发的基于微信号结构的高性能数字信号处理器，非常适合于各种音频、视频、通信及复杂控制等领域，市场占有率也名列前茅。

本书从实用性的角度出发，首先介绍了数字视频处理的基本概念和典型算法，以及 Blackfin DSP 的基础知识和基本使用方法。然后就视频处理系统的硬件搭建和软件开发进行详细介绍，实现了一个包括视频输入、输出和传输等基本功能的最小视频系统，作为进一步开发复杂系统的基础。接下来讨论了视频开发过程中应遵循的设计原则，提供了典型的参考开发模板。考虑到复杂图像处理系统的开发已经进入模块化、集成化的开发阶段，充分利用已有的、经过优化的开发包则是快速、高效开发高性能视频处理产品的必然趋势，本书最后部分结合 ADI 公司提供的 Blackfin 图像和视频开发包（类似于 OpenCV），介绍了运动分析、视频编解码、视频滤波的基本原理，以及在 Blackfin 处理器上的实现方法。

本书共 9 章。第 1 章是绪论，介绍数字视频处理的概念、历史、应用和主要研究内容。第 2 章介绍数字视频处理的基本理论和处理模型。第 3 章介绍 Blackfin DSP 的结构、特性和工作原理，还讨论了 DSP 应用开发的一般过程和集成开发环境 VisualDSP++的基本使用。第 4 章是基于 Blackfin 处理器的最小视频系统，实现了视频的输入、输出和数据传输，为更复杂的处理系统的开发打好了基础。第 5 章讨论了视频系统的设计原则，提供了典型的视频开发模板，并用具体例子进行了说明。第 6 章介绍 Blackfin 图像与视频处理开发工具包，它提供了丰富的软件模块，能够显著加速学习和开发过程，文中还介绍了一些典型模块的使用。第 7 章视频运动分析及应用，介绍运动估算、运动分割和运动跟踪的理论和算法，以及基于图像视频开发包的实现方法。第 8 章介绍视频编解码的基本理论和算法，并针对 H.264 基线类介绍了基于 Blackfin 的代码移植和优化策略，并详细介绍了 Blackfin 提供的 H.264 编解码器的使用方法。第 9 章介绍视频时域和空域滤波的基本理论，并介绍了基于运动检测的自适应去交错效应的实现算法。

本书的编写得到了 ADI 公司的大力支持，为本书提供了丰富的文档和网络资源。邢延超主要负责第 1 章至第 3 章及第 7 章至第 9 章的撰写；皇甫伟主要负责第 4 章至第 6 章的撰写。在材料整理、录入和校对方面，王治中、庞秀娟、王清贵等做了大量的工作。另外，本书的完成更离不开家庭作为坚强的后盾。在此一并向他们表示衷心感谢！

由于本人水平有限，书中错误之处在所难免，恳请广大读者批评指正。

编著者

2011 年 4 月

序　言

这些年，在与电子技术领域的工程师、学者以及大学师生交流的时候，他们的聪明才智和创新能力给我留下了深刻的印象。而他们所做的设计和项目，无一不让我感觉到中国工程师队伍成长之快，和中国电子行业巨大的发展潜力。但另一方面，他们的经历和成功，也带给了我很多思考。

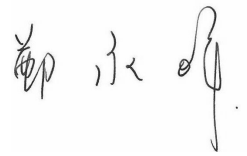
ADI 在模拟和数字信号领域中已经发展了 40 多年。在这几十年间，我们不断推动技术的创新和进步，不断提高相关领域的各类产品性能以满足客户的广泛需求，包括消费类、通信、医疗、运输和工业等方面。令人欣慰的是，至 2009 年，ADI 已经拥有遍布世界各地的 60,000 余家客户。而通过大学计划、培训、研讨会等活动所积累起来的资源更是不计其数。如何让我们的客户，让 ADI 技术产品的使用者和爱好者，真正准确、有效、快捷地掌握相关知识与设计技巧，是我们需要考虑的，也是我们为所有用户提供的非常重要的服务之一。

经过多年的运行和完善，ADI 已经拥有了一整套对中国工程师以及在校工科类学生的培养计划，如每年一届的中国大学创新设计竞赛，在高校建立的联合实验室，各类线上线下研讨会，还有在多个城市开展的高水平培训课程等等。这些计划架起了 ADI 与用户之间最直接、最有效的沟通桥梁。同时，为了使更多的电子技术领域从业者和爱好者了解数字信号处理和电子产品设计理念，我们还邀请了业内具有较深影响力的专家、学者、教授共同编写并出版一套基于 ADI 模拟和数字产品的应用技术丛书。

该丛书详细介绍了 ADI 产品在医疗电子，通信，工业仪器仪表，汽车电子等行业的应用，以理论与实际案例相结合的方式为读者们讲解了世界先进处理器的设计与使用。

丛书的出版凝聚了来自清华大学、西安电子科技大学、青岛理工大学、解放军理工大学、厦门大学、天津大学、黑龙江大学、中国科学技术大学、辽宁工业大学等多所院校老师丰富的经验和智慧。在此，感谢他们对 ADI 出版计划的大力支持。同时，也感谢电子工业出版社的竺南直博士对本丛书的出版所作出的贡献！

衷心希望能得到读者朋友的意见反馈，在你们提出的问题和建议下，我们将不断完善 ADI 丛书，不断完善 ADI 的产品和技术，与客户们一起共同开拓中国市场。



ADI 公司亚太区副总裁

目　录

第 1 章　绪　　论

1.1　数字视频处理概述

众所周知，视觉是人类感知信息最重要的途径，人类从外部世界获取的信息 70%～80%是通过视觉获取的。随着电子信息技术的发展，人们已经能够利用各种电子设备完成视频信息的采集、编解码、存储、传输和处理等操作，为高效地分析和处理客观世界提供了丰富的手段。其中模拟视频信号已经应用了几十年，至今仍在使用，最原始也是最常见的通用视频标准包括 NTSC、PAL 和 SECAM 等。其他的现代消费模拟视频传输系统包括 S-Video、分量视频等系统。现代视频系统通常在前端和后端分别采用模数转换和数模转换技术，在系统内部采用数字信号处理技术来对视频内容进行处理，大大提高了系统的灵活性，并具有相应的低成本及集成特性，因而数字化产品更吸引消费者。随着电子、通信、计算机等技术的飞速发展，数字视频产品性能不断提高，同时价格也能够为大众所接受，因而具有广大的潜在市场。其中，基于数字信号处理器（DSP）的嵌入式多媒体系统由于具有体积小、功耗小、成本低等优点，使得该类产品越来越实用。从事数字视频处理系统和产品的研发人员不仅需要掌握数字视频处理的基本理论、方法和技术，同时还要了解嵌入式系统软硬件开发平台的相关知识。本书正是从这两个方面出发，本着实用化的思想来介绍基本开发过程和技术，希望能够为读者尽快掌握基于 DSP 平台开发视频处理系统和产品提供帮助。

视频信息分为图像和视频两大类，前者是静态的，后者是动态的。与静态的图像相比，视频信息无论是在存储空间需求、传输带宽需求，还是在处理的实时性上，都对整个处理系统提出了更高的要求。在数字信号处理应用的早期，由于存储、带宽及处理器处理能力的限制，人们在谈到数字信号处理时，更多关注的是语音和图像信号的处理。随着电子、通信、计算机等相关技术的飞速发展，上述三方面的限制在很大程度上得到了解决或改善，又由于视频信号处理对社会和经济发展具有重大的推动作用，人们开始积极地展开了对视频信号处理的研究与应用。

视频信号本身是模拟的，传统的广播电视信号就是一种典型的模拟视频信号，它由摄像机通过电子扫描将随时间和空间变化的景物进行光电转换后，得到一维的时间函数的电信号，其电平的高低反映了景物的色彩值。模拟视频信号在传输、存储、处理和交互操作等方面具有很大的局限性。为此，可以将视频信号数字化，得到数字视频信号。数字视频信号便于传输，便于多媒体通信，便于存储、处理和加密，无噪声累积，便于设备的小型化。随着数字电路和微电子技术的进步，特别是超大规模集成电路的快速发展，使得数字视频信号的优点越来越突出，应用越来越广泛。例如，高清晰度电视（HDTV）、多媒体、

视频会议、移动视频、监控系统、医疗设备、航空航天、教育、电影等。

数字视频信号处理系统主要包括视频信号的采集、数字化、视频编解码、存储、处理、传输、回放等主要模块。每项功能都由相应的设备来完成，如摄像机负责视频信号的采集，模数转换器负责信号的数字化，编解码完成视频编解码等。广义地讲，上述内容都属于数字视频处理的范畴。与之相对应，狭义的数字视频信号处理主要指，对已数字化的视频信号进行某种特殊功能的分析和加工，如帧间滤波、视频压缩、图像增强、运动目标跟踪及动态场景分析等。但是与语音信号和图像信号的数字化处理相比，数字视频信号数据量巨大，而且对处理的实时性要求高。通用微处理器在实现数字信号处理上效率不高，一般难以满足数字视频处理的实时性需求，因此产生了专门针对数字信号处理的微处理器 DSP（Digital Signal Processor，数字信号处理器）。

本书将介绍数字视频处理系统的各个环节的理论和系统实现，重点介绍基于 DSP 的各种算法的实现。

1.2 数字视频的发展历史

数字视频的发展历史与计算机的发展史、与计算机能够处理的信息类型密切相关。自 20 世纪 40 年代计算机诞生以来，计算机大约经历了以下几个发展阶段：数值计算阶段和多媒体阶段。

数值计算阶段是计算机发展的初级阶段，这个时期它只能处理数值数据，主要用于解决科学计算与工程计算中的数值计算问题。例如，世界上第一台电子计算机 ENIAC 就是为美国国防部解决弹道计算问题、编制射击表而产生的。

接下来是多媒体阶段。随着电子技术的发展，各种支持图形、图像和语音的设备的问世，计算机逐渐进入多媒体时代，处理的信息载体扩展到文本、图形、图像、声音等多种类型，这不仅扩大了计算机的应用领域，同时也极大地改善了人机接口的友好性。由于视觉信息（如图形、图像）能够直观明了、生动形象地传达有关对象的信息，因而在多媒体计算机中占有重要的地位。

进入多媒体阶段后，基于计算机系统的视频信息的获取、处理、存储、显示等得到了广泛认可和应用。由于普遍使用的是数字计算机，其内核只能处理数字信号，所以我们研究的是数字视频处理技术。数字视频的发展大致可以分为初级阶段、主流阶段和高级阶段。

初级阶段主要是在微型计算机上增加简单的视频功能，利用计算机来处理活动画面。但此时由于设备尚未普及，一般都是面向视频制作领域的专业人员，普通 PC 用户还无法奢望在自己的计算机上实现视频功能。

在初级阶段，数字视频的发展比较缓慢，其主要原因是数字视频的处理、存储和传输能力都不足，无法胜任所需要的处理、存储和传输能力。例如，一分钟的真彩色数字视频需要 1.5GB 的存储空间，而早期一般台式机配备的硬盘容量大约是几百兆，显然无法胜任如此大的数据量。

尽管存在这些限制，人们还是采用一些折中方法来对数字视频进行处理。比如用计算机捕获单幅视频图像，将其以一定的文件格式存储起来，利用图像处理软件进行处理，将结果保存下来用于需要的各种场合。后来，又实现了在计算机上显示活动的视频，虽然画面时断时续而且持续时间很短，但毕竟使图像动了起来，给人们带来了崭新的体验。

后来，随着计算机软硬件性能的不断提高，以及视频采集设备、大容量存储设备、视频显示设备等不断升级，最终使得视频捕获、存储、播放在个人台式机上成为可能。由此进入了数字视频处理的主流阶段，即模拟视频不再是视频处理的主流。这其中非常关键的是压缩解压缩（Codec）技术的成熟，压缩可以极大地降低数据量（达数十上百倍）。压缩的实现又分为纯软件压缩和硬件辅助压缩两种，前者方便易行，成本低，但是速度较慢，难以满足实时性的要求；后者速度快，但是成本高。如果尺寸较小、帧速较慢，通过视频压缩后 1 分钟的视频数据只需 20MB 而不再是 1.5GB，压缩比高达 1∶75。这样就能够将视频文件保存在硬盘上，也可以进行动态的显示，虽然分辨率仅为 160×120，帧速率只有 15 帧/秒，色彩也只有 256 色，但画面毕竟活动起来了。这样数字视频处理就不再是专业人员的特权，而成为个人计算机的必备功能。

高级阶段又称成熟的多媒体计算机阶段，各种计算机外设产品日益齐备，数字影像设备争奇斗艳，视音频处理硬件与软件技术高度发达，这些都为数字视频的流行起到了推动的作用。在这个阶段，数字视频被进一步地标准化。

在数字视频的编解码方面。首先，为了 PAL、NTSC 和 SECAM 电视制式之间确定共同的数字化参数，国际无线电咨询委员会（CCIR）制定了广播级质量的数字电视标准，称为 CCIR 601 标准。在该标准中，对采样速率进展迅速，包括图像尺寸、帧率、采样结构、颜色空间选择等都做了严格的规定。这种未压缩的数字视频数据量对目前的计算机和网络来说是无法实现的，因此需要对其进行数字压缩。这就要求对压缩算法和存储格式进行标准化，现在普遍采用的压缩标准包括 MPEG 系列的 MPEG-1、MPEG-2 和 MPEG-4，以及 H 系列的 H.261、H.263 和 H.264 等。这些标准的确定使不同系统之间可以交换和共享数字视频内容。

视频内容数字化以后，就可以采用（二维）数字信号处理技术进行灵活地处理，如进行视频滤波、图像增强、图像缩放、运动估计、目标检测与跟踪等。数字视频处理的应用不断丰富，如智能视频监控、视频增强、视频滤波等。当前数字视频处理的理论和技术成为研究的前沿和热点，同时必将不断拓宽视频处理应用的范围。

1.3　数字视频处理的应用

数字视频在监控、无线视频网关和数码相机等方面有广泛的应用，这些应用的共同点是它们都要求视频和图像的处理。这些处理可能是视频图像质量的优化，如锐化和白平衡，也可能是视频和图像的压缩/解压缩，如 MPEG4 或 H.264 等。主要的视频应用包括以下几类。

1．数字电视机顶盒

机顶盒负责在用户端同步解码、处理和播放视频、音频和数据流，有些机顶盒还支持对 DVB 或 DVD 其他 MPEG 数据流的编码功能。这些机顶盒一般功能比较简单，主要是负责接收数据流然后解码或编码并进行显示。但是由于是进行流处理，并且多半是 MPEG2 质量的视频流，所以需要处理能力较强。

2．数字视频监控系统

数字视频监控系统相对传统视频监控系统来说是一个显著的飞跃，它增加了更多的智能特性，如可以进行移动报警，通过视频处理方法在监控区域发现移动物体就报警；数字化的视频档案更加便于存储、管理和远程传输，如可以通过自动视频分析方法来找到目的片段，数字化信息通过网络更便于传输。数字视频监控系统最主要的要求是可编程性，数字处理能力也是实现这些特点的保证。另外，视频输入和网络接口也不可或缺。

3．IP 视频电话/视频会议

IP 视频电话是将实时的多通道视频集成到现有的 IP 语音电话中，视频会议则是在多点间实现视频电话会议。目前，这两种应用多采用 H.263 或 MPEG4，芯片要完成这两种制式的编解码，同时完成图像采集处理的任务，单芯片处理能力要求较强。由于要进行视频采集，因此接口要求支持视频输入，要实现 TCP/IP 协议则要求有以太网接口的输出支持。

4．个人数字视频播放器/点播机

其主要任务是完成视频解码，前者更强调便携性，而后者则强调丰富的节目源。在该应用中，处理性能并不是第一位要考虑的因素，而整个系统的低能耗十分关键。另外，要求芯片的接口比较丰富以支持 LCD 显示屏、CFC 或其他存储卡或硬盘接口。

5．无线视频网关

无线视频网关位于 MSC 内，用于将任何制式的视频内容转成手机支持的格式，从而将这些内容在无线网络上发送。由于无线网络对环境的依赖比较强，不同连接的信道质量会有所不同，因此要求网关针对信道质量，将视频内容转换到该信道所支持的码率上。由于网关要与其他网络接口，要求芯片的外围接口支持如 ATM 或以太网等网络总线。

除此以外，还有各种丰富的数字视频处理系统，如数字摄录机、数字照相机等。这些应用总体上分为两大类：一类是高性能，要求多通道处理或多编码制式，或追求高图像质量，同时由于技术的不断发展，它又需要较强的可编程性支持未来的技术发展；另一类是便携应用，强调小型化、低功耗，并能支持多种移动存储接口及 USB、1394 或其他高速总线。

1.4　数字视频处理的研究内容

尽管数字视频系统多种多样，处理目的各不相同，但视频处理的基本内容和相关技术可以归纳为以下几个方面[13~17,18~21,23~25]。

1．视频信号表示与标准化

模拟电视系统通常用光栅扫描方式，即在一定的时间间隔内电子束以从左到右、从上到下的方式扫描采集荧光屏表面。扫描方式通常有逐行扫描和隔行扫描两种。经行场扫描得到的一维连续时间信号包括图像信号、行同步信号、行消隐信号、场同步信号、场消隐信号和前后均衡脉冲等。视频信号的基本参数主要有清晰度、分解力、宽高比、行频、场频和帧频等。模拟电视系统标准主要有 NTSC 制、PAL 制和 SECAM 制，它们对上述各种参数作出了不同的标准化，方便了视频设备的兼容和视频内容的交互。

模拟视频信号经过数字化处理后，就变成由一帧帧数字图像组成的图像序列，即数字视频信号。每帧图像由 N 行、每行 M 个像素组成，每个像素用 N_b 比特表示。为了便于国际节目交换及 625 行 PAL 制系统与 NTSC 制系统之间的兼容，1982 年 CCIR（国际无线电通信咨询委员会）制定了 CCIR 601 数字视频标准，1993 年变更为国际电信联盟无线通信部门 ITU-T BT.601 建议，定义了对应于 525 行和 625 行电视演播室的数字编码参数。三个分量信号 Y、R-Y、B-Y 的抽样频率分别为 13.5MHz、6.75MHz 和 6.75MHz，三个分量在一行中抽样点数的比例为 4∶2∶2，且每帧的行数相同，故称为 4∶2∶2 标准，此外还有图像质量更高的 4∶4∶4 和较低图像质量的 4∶1∶1 标准或 4∶2∶0 标准。

2．二维运动估计

建立在数字图像处理技术的基础上，数字视频处理主要关注对视觉信息动态部分的分析、处理和利用。运动分析广泛应用于计算机视觉、目标跟踪、视频监视、视频压缩等应用场合。不同应用场合对运动估计的要求不同，有时强调估计结果与实际运动尽可能一致，称为真实运动分析；有时则强调获取更高压缩率，不要求估计结果与实际运动的对应关系。

运动估计处理的对象是视频图像序列，是三维场景在图像平面上的投影。在不同时刻拍摄到的投影图像上，运动物体的投影坐标是不同的。运动分析与估计的研究内容就是测量计算投影坐标在图像平面上的变化，并用来分析运动物体的结构，估计物体的运动参数。由于摄像机的投影过程是不可逆的，投影过程不可避免地要丢失一些信息，因此估计物体的真实运动和结构是非常困难的。

二维运动估计的基本问题是估计运动前后相邻时刻两幅图像上对应点的坐标，即二维运动矢量。三维运动估计的基本问题是利用一组对应点坐标估计和确定运动物体的结构和运动参数。通常运动估计算法均假设物体点的亮度（或颜色）在其运动轨迹上保持不变。目前，运动分析方法主要有两种：一是，根据时间相邻的两幅或多幅图像求解物体的运动

参数和三维结构信息；二是，图像序列的光流分析方法。

二维运动估计方法主要有以下几类：

（1）基于光流的运动估计，利用光流场和各种约束条件求解各个位置上的运动矢量。

（2）基于像素的运动估计，包括位移帧差、多点邻域约束、像素递归法、基于贝叶斯准则的方法等。

（3）基于块的运动估计，它假定每个块只做二维平移运动，已成功应用于各种视频压缩编码标准。

（4）基于网格的运动估计，利用网格的形变体现物体的形状变化。

（5）基于区域的运动估计，将视频图像分割为多个区域，每个区域对应一个特定的运动，然后可以为每个区域估计运动参数。

（6）多分辨率运动估计，利用视频图像的多分辨率原理，首先在最小分辨率层进行运动估计，然后逐层进行，最后得到最高分辨率的运动场。

3．三维运动估计

三维运动估计广泛应用于机器视觉、视频监控、交通控制等场合。三维运动估计的基本任务是分析和估计三维场景中物体运动的情况。由于物体的结构与投影结果密切相关，所以同时需要估计三维物体的结构参数。由于很难精确估计出物体的运动结构，因此需要作出一些合理的假设来简化运动模型，如刚体运动限制等。三维运动和结构的估计方法可以分为两类：一是，间接估计法，根据已经给出的二维运动矢量（如特征点对、光流场）来估计二维运动和结构参数；二是，直接估计法，根据视频图像的空时亮度信息来估计三维运动和结构。

三维运动估计主要有以下几种方法：

（1）基于特征对应的运动估计，属于间接运动估计方法，如最小二乘估计方法。

（2）基于光路的运动估计，也属于间接运动估计方法，又分为正交投影下的运动估计和透视投影下的运动估计，首先估计二维图像的光流场，然后根据光流和光流参数模型估计三维运动参数和结构参数。

（3）直接运动估计法，如用图像梯度代替光流矢量，然后基于平面运动模型直接估计运动参数的方法等。

4．运动目标分割

三维场景中通常包含多个运动物体，运动分割就是将视频序列中属于各个不同运动的像素标记出来。运动分割与运动估计密切相关，精确的运动分割可以提高运动估计的准确性，反之亦然。在基于内容的视频描述中，为了得到描述视频的每个对象，需要通过运动分割将运动物体与背景分割开来。运动分割的主要方法包括基于空时图像的直接分割方法、基于光流场参数模型的分割方法及运动分割和运动估计同时进行的方法。

直接分割方法属于分割优先的方法，首先基于图像亮度信息进行区域分割，然后估计每个区域的运动参数。基于光流的分割法属于运动优先的方法，首先估计光流场，然后对

光流场建模并进行运动分割。最常见的光流场参数模型有 6 参数仿射流模型和 8 参数二次流模型。运动分割可以通过将具有相同模型的参数矢量归为一类得到。运动分割和运动估计同时进行的方法，由于运动分割和运动估计是相互依赖和相互促进的，因此同时进行运动估计和运动分割可以得到更好的结果。

5．运动目标跟踪

运动跟踪的基本任务是通过对摄像机拍摄到的图像序列进行分析，计算出目标物体在各图像帧上的位置，并给出物体运动的估计。运动跟踪通常基于一组特征对应或光流矢量来描述运动的发展过程，其中物体特征或光流矢量称为运动跟踪的观察对象。运动跟踪主要包括三个方面的内容：运动模型、观察模型和跟踪方法。

运动模型可以是简单的匀速运动模型，也可以是更加复杂的存在旋转、形变等的模型。运动模型又可以分为二维运动模型和三维运动模型，一般主要研究刚体运动模型。观察模型可以是一组特征对应关系或光流矢量。跟踪方法利用运动模型和观察模型估计最佳的运动模型参数，分为两类：一类是分批法，如非线性最小二乘法，基本思想是在获取一定的观察数据后对现有全部数据进行处理；另一类是递归法，如 Kalman 滤波器等，只要获取的数据可以更新运动参数，递归算法就能很好地工作，通过逐步迭代得到理想的跟踪结果。

6．视频滤波

在视频采样与数字化过程中，由于设备噪声、采样方式转换、隔行扫描模式、场景运动等因素往往造成视频质量的下降，有时则是希望利用现有视频信号重构超分辨率的高清晰度视频信号。这都属于视频滤波研究的内容，它具体又包括运动补偿滤波、噪声滤波、图像复原、上下行变换、超分辨率等内容。各个研究内容有各种丰富的处理技术，如噪声滤波就有帧内滤波、运动自适应滤波和运动补偿滤波等不同种类的技术。

7．视频压缩编码

视频原始数据量巨大，难以直接进行存储和传输，因此数据压缩具有特别重要的意义。视频压缩主要采用有损压缩方式，与静态图像压缩相比，视频信号压缩可以充分利用帧间相关性去除时间冗余度。因此视频压缩编码又分为帧内压缩和帧间压缩两部分，帧内压缩与静态图像压缩类似，主要利用变换编码（如 DCT）来实现。帧间压缩则通过计算图像块的运动矢量，利用参考帧信息来重构预测帧的信息，因而能够极大地提高压缩效率。现在通用的 MPEG-2、H.264 等编码标准普遍采用这一思想。

近些年来则提出了基于模型的编码技术，它通过将场景分割为相互独立的不同目标来进行单独编码，在显示端重构出整个场景，不仅能够取得更高的编码效率，同时也为交互式视频提供了基础。

数字视频处理的内容还有很多，如三维立体视频、数字视频水印技术等，这里就不一一介绍了。

1.5 数字视频处理系统概述

数字视频处理系统组成，如图 1.1 所示。

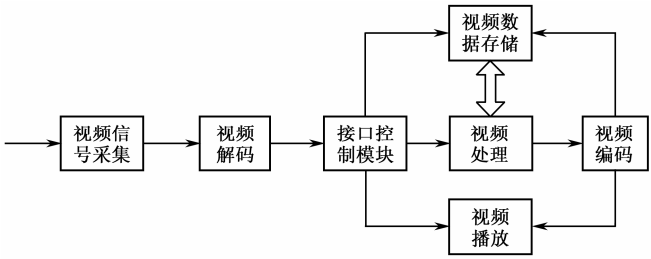


图 1.1 数字视频处理系统组成

数字视频处理系统的第一个模块是视频采集模块，它负责将光信号转化为电信号。为了实现数字视频信号处理，需要对视频模拟信号进行视频解码，这可以用视频采集卡来实现。数字化的视频数据需要进行存储、处理和播放，接口控制模块负责控制视频存储模块、视频处理模块、视频编码模块和视频播放模块的工作，以及它们对存储的视频数据的访问。

1.5.1 视频信号采集[15,22]

视频信号采集的常用工具包括录像机或摄像机等。摄像机种类繁多，其工作的基本原理都是一样的：把光学图像信号转变为电信号，以便于存储或传输。当我们拍摄一个物体时，此物体上反射的光被摄像机镜头收集，使其聚焦在摄像器件的受光面（如摄像管的靶面）上，再通过摄像器件把光信号转变为电信号，即得到了“视频信号”。光电信号很微弱，需通过预放电路进行放大，再经过各种电路进行处理和调整，最后得到的标准信号可以送到录像机等记录媒介上记录下来，或通过传播系统传播或送到监视器上显示出来。

摄像器件分摄像管和固体（半导体）摄像器件两大类。摄像管器件又分为析像管、光电倍增析像管、超正析像管和光导摄像管等几种。新型摄像机中多使用小巧的氧化铅光电摄像管。各种摄像管都有一个真空玻壳，里面装有靶面和电子枪。被摄景物透过玻壳上的窗成像于靶面，利用靶面的光电发射效应或光电导效应将靶面各点的照度分布转化为相应的电位分布，将光图像变成电图像。在管外偏转线圈驱动下，电子束逐点逐行扫描靶面，把扫描路径上各像素的电位信号按序输出。固体摄像器件是一种新型的电荷耦合器件(CCD)。几十万个器件单元排列成阵面，表层具有光敏特性。被摄景物成像于阵面，各单元存储电荷量和照度成正比。利用时钟脉冲和移位控制信号，将阵面各单元信号按一定顺序移出，即可得到强度随时间变化的图像电信号。

摄像机的主要性能指标包括信噪比、最低照度、灵敏度、解析力、几何失真和重合失真等，它们从不同方面影响着视频信号的质量。可以通过后期视频信号处理算法来减轻或

消除某些因素的影响，提高视频信号的质量，如几何失真等。按照质量等级分类，可以分为广播级摄像机、业务级摄像机、家用级摄像机等（如图 1.2 所示），它们对清晰度、信噪比有不同的要求，同时在体积、功能、操作复杂度和价格上具有明显的差别，适应于不同的应用场合。

此外还有红外摄像机，红外摄像技术分为被动红外和主动红外两种。被动红外摄像技术是利用任何物体在绝对零度（−273℃）以上都有红外光发射的原理。由于人的身体和发热物体发出的红外光较强，其他非发热物体发出的红光很微弱，因此，利用特殊的红外摄像机就可以实现夜间监控。被动红外摄像技术由于设备造价高且不能反映周围环境状况，因此在夜视系统中不被采用。主动红外摄像技术是利用特制的“红外灯”人为产生红外辐射，产生人眼看不见而普通摄像机能捕捉到的红外光，辐射“照明”景物和环境，利用普通低照度 CCD 黑白摄像机或使用白天彩色夜间自动变黑白的摄像机或红外低照度彩色摄像机去感受周围环境反射回来的红外光，从而实现夜视功能。

图 1.2　各种不同种类的摄像机

为了实现数字视频信号处理，需要对视频模拟信号进行视频解码，这可以利用视频采集卡实现。视频采集卡的主要功能是从动态视频中实时或非实时捕获图像并存储。通过视频源与视频解码设备的融合，数字视频源的应用逐渐普及，这进一步简化了视频处理系统的构成。

第一台数码摄像机诞生于 1995 年，在这十几年中，数码摄像机发生了巨大变化：存储介质从 DV 到 DVD 再到硬盘，总像素从 80 万到超过 1000 万，影像质量从标清 DV 到高清 HDV。常用的传感器类型有 CMOS 与 CCD。CCD 使用一种高感光度的半导体材料制成，能把光线转变成电荷，通过模数转换器芯片转换成数字信号。CMOS 即互补性氧化金属半导体，与 CCD 一样同为在数码摄像机中可记录光线变化的半导体。在相同分辨率下，CMOS 的价格比 CCD 便宜，但是 CMOS 器件产生的图像质量相比 CCD 来说要低一些。CMOS 感应器则作为低端产品应用于一些摄像头上，不过一些高端产品也采用了特制的 CMOS 作为光感器。

1.5.2 数字视频编解码[20,21]

采集到的视频信号需要转换成标准格式才能进行有效地传输和播放。视频采集卡既可以采集模拟视频，如摄像机、摄像头、录像机、DVD 机等模拟视频源，对信号进行采样、量化后转化成数字信号，然后压缩编码成数字视频；也可以采集数字视频。数字视频采集是指利用可连接 DV 视频信号的 IEEE 1394 接口，完成将数码摄像机拍摄的 DV 信号采集到多媒体计算机系统的功能。

大多数视频卡都具备硬件压缩的功能，在采集视频信号时首先在卡上对视频信号进行压缩，然后再通过 PCI 接口把压缩的视频数据传送到主机上。压缩标准包括 AVI、MPEG-1、H.263、MPEG-2、H.264 等。为了做到实时采集，视频采集卡必须在采集下一帧图像之前完成本帧图像的处理并传入 PC 系统。如果每帧视频图像的处理时间超过相邻两帧之间的相隔时间，则要出现数据的丢失，即丢帧现象。通过视频采集卡，可以把视频信号转存到计算机中，再利用相关的视频编辑软件，对数字化的视频信号进行后期编辑处理，如剪切画面，添加滤镜、字幕和音效，设置转场效果及加入各种视频特效等，最后将编辑完成的视频信号转换成标准的 VCD、DVD 及网上流媒体等格式，方便传播和保存。

随着技术的发展，今天的多数数字摄像头都可以通过内部电路直接把图像转换成数字信号传送到计算机上。只要 CPU 处理能力足够快，CCD 捕捉到的图像信号基本可以达到实时的动态效果。数字摄像头的连接方式基本通过三种方式实现：接口卡、并口和 USB 接口。接口卡方式一般是通过摄像头专用卡来实现，厂商多会针对摄像头优化或添加视频捕获功能，在图像画质和视频流的捕获方面具有较大的优势，但由于各厂商的接口卡的设计各不相同，产品之间无法通用，加上价钱也不便宜，这类产品值得需要追求较高画质的用户选择。并口方式的优点在于适应性较强，每台机器都有并口，不过数据传输率较慢，实用性大为降低，但对于普通用户来说，还是可以接受的。USB 接口方式是目前主流的走向，现有的主板都支持 USB 连接方式，方便和强大的扩充能力是 USB 接口的最大优点，而且现在的数字摄像头的功耗较小，依靠 USB 提供的电源就可以工作，这样可以省去外接电源。

IEEE 1394 总线是一种高速串行总线，支持的传输速率有 100Mbps、200Mbps、400Mbps，将来会提升到 800Mbps、1Gbps 和 1.6Gbps。它不需要控制器，可以实现对等传输，最大连线 4.5m，大于 4.5m 可采用中继设备支持，同样支持即插即用。火线是目前唯一支持数字摄录机的总线。IEEE 1394 既可作为外部总线，又可作为内部总线使用。现在支持 IEEE 1394 的设备不太多，只有数码相机等一些使用高带宽的设备使用 IEEE 1394。其他设备其实也用不了那么高的带宽。IEEE 1394 总线需要占用大量的资源，所以需要高速度的 CPU。

1.5.3 数字视频存储

视频数据量非常大，对存储设备要求很高。视频存储设备分为内置存储和外置存储两大类，外置存储又分为直连式存储和网络存储两类。视频存储设备类型之一的直连存储依

赖服务器主机操作系统进行数据的 IO 读写和存储维护管理，数据备份和恢复要求占用服务器主机资源（包括 CPU、系统 IO 等），数据流需要回流主机再到服务器连接着的磁带机（库），数据备份通常占用服务器主机资源 20%～30%，直连式存储的数据量越大，备份和恢复的时间就越长，对服务器硬件的依赖性和影响就越大。

存储网络可以分为 NAS（Network Attached Storage，网络接入存储）和 SAN（Storage Area Networks，存储区域网络）。NAS 用户通过 TCP/IP 协议访问数据，采用业界标准文件共享协议，如 NFS、HTTP、CIFS 实现共享。NAS 每个应用服务器通过网络共享协议（如 NFS、CIFS）使用同一个文件管理系统。SAN 通过专用光纤通道交换机访问数据，采用 SCSI、FC-AL 接口。NAS 和 SAN 最本质的不同就是文件管理系统（FS）放置位置的不同。在 SAN 结构中，文件管理系统分别放在每一个应用服务器上。NAS 系统则有自己单独的文件管理系统。

随着便携式电子设备在全球的风行，广泛应用于手机、数码相机、闪存盘、便携式音乐播放器的闪存市场正在突飞猛进。NOR、NAND 及新近出现的 ORNAND 闪存市场增长迅猛。NAND 闪存是一种面向数据块的大量存取装置，它能够为数据和代码存储提供更低的单位比特成本。NOR 则是具有字节寻址能力的设备，其典型应用为代码存储。NAND 具有更低的功率耗散及更高的数据块传输速度。NOR 则具有更短的延迟时间，该技术针对字节传输及代码存储应用进行了优化。因此高速度和低成本分别是 NOR 和 NAND 闪存的最大特点，分别适用于手机或 MP3 等。

HDD 硬盘和闪存的关系是既竞争又合作。两者都在不断发展，其基本差异体现在耗电量及外形这两大因素。闪存比 HDD 拥有更小的尺寸、更低的功耗。此外，闪存是一种固态器件，没有 HDD 所必需的机械移动部件，因此它对便携式应用具有更好的可靠性。虽然在需要小容量的数字内容装载应用上，闪存是一个不错的解决方案，但有限的数据传输率和数据写入次数是闪存面临的两大技术问题。因此在需要高质量数据流的影像播放存储场合 HDD 就具有很大优势。闪存技术的发展可以带来更多的便携式数字移动设备，同时可以带来更多的存储应用。低成本、低容量的应用是闪存为主的市场，而高容量、高数据流的应用需求是以 HDD 为主。

1.5.4　数字视频显示[15]

目前，主要的显示器件有 CRT、AMLCD、PMLCD、ELD（场致发光显示器）、FED（场发射显示器）、VFD（真空荧光显示器）、OLED 和 PDP 等。除 CRT 和投影显示器外，其他显示器件都属于平板显示器件（FPD）。

在诸多显示器中，CRT（射线管）发展历史最久，技术最成熟。它的优点包括：性价比高；很容易调整分辨率（VGA～VXGA 和 HDTV）；形状和大小变化范围大（0.5～45 英寸）；寻址简单（只有 7 根导线）；可视性好（高亮度和高对比度）；具有非常好的发光效率等。其弱点包括：大屏幕产品笨重；屏面内有光散射；图像有闪烁、抖动和畸变；存在视觉疲劳和电磁辐射；最大直观显示尺寸被限于 45 英寸以下；无数位寻址；工作电压和功耗较高；

在荫罩彩管内分辨率受到限制。

液晶显示（LCD）器件包括有源矩阵（AM LCD）和无源矩阵（PM LCD）两种。AM LCD 的主要特点是：高性能、彩色、高分辨率、快速、轻薄、有市场。它主要应用于个人视频用品（TV 等）、笔记本计算机和桌上监视器等。PM LCD 价格便宜，性能良好，有一定的市场，供货商很多。它主要应用于消费类电子产品和通信产品，在低功率、单色、低性能等应用领域占有优势。PM LCD 的性能虽然不是最好的，但能满足大规模市场的需求。LCD 功率低、薄、轻，做成 IC 驱动器可使平板更薄，可用于携带型产品、笔记本计算机、监视器、手机等，市场应用范围较广泛。但是它的制作工艺复杂，需要严格控制；主要设备价格昂贵；生产成本比较高；视角窄；制造米级的大屏幕显示屏很困难，且成本十分昂贵。

等离子体显示（PDP）的主要特点是大屏幕、全彩色和视频显示。它的主要应用领域是公共场所信息显示、广告、电视和 HDTV 等。PDP 的优点是：大屏幕显示（达 63 英寸）、宽视角、薄、对比度好、亮度高（可达 600cd/m^2）、响应时间快、有众多的供货商提供支持、可用于 TV。但是，目前 PDP 的显示板和电子电路成本仍然很高，功耗大，目前普遍采用的工艺不可能简化，制造工艺也不成熟，在高像素密度（高分辨率）下发光效率低。

有机发光二极管（OLED）的主要特点是低电压（5～20V）、低功耗、自发光、高亮度、高对比度、薄、轻（重量只有 LCD 的一半）、全彩色显示、高发光效率、快速响应、宽视角、单片结构、加工不复杂、成本低。它主要应用于数字摄像机、PDA、手提式产品等消费类产品，及汽车显示器、头盔式显示、计算机显示器、视频显示等工业类产品。无源矩阵 OLED 适合于文字显示，有源矩阵 OLED 适合于视频及图表显示。预计 OLED 将会在许多应用领域挤占 LCD 的市场。

场发射显示板（FED）被认为是 CRT 的最好继承者，它具有 CRT 的优点，又克服了 CRT 体积笨重的缺点，而且功耗较低。但是其结构复杂，封装困难，寿命问题还未解决，目前市场还不成熟。一旦寿命和制造问题得到解决，FED 将会成为新一代的显示器。

未来的显示器件将发生很大的变化。HDTV 对显示器件的要求是：显示多屏幕图像、无几何畸变、全平面聚焦、TV 屏幕应不小于 50 英寸、纯平面屏幕的薄型显示器、图像质量/颜色再现应优于 CRT、必须是自发光型，显示屏尺寸应覆盖 10～50 英寸范围，并且生产成本应低于 CRT。目前，还没有一种显示器件能完全满足要求。从现有显示器的状况来看，PDP 和有机 EL 有可能满足要求。

未来显示器应具有纸张的优点——柔软、可以卷折、便于携带，同时又可以显示和记录动态的文字和图像。这种显示器还可以像纸张一样装订成“书”，形成多页显示器。显示器的电源可以是太阳能电池，驱动电路可以由有机晶体管集成电路组成。其次，未来显示器应该由显示平面图像发展成显示立体图像。立体显示技术将是未来发展显示技术的一个很重要的方向。21 世纪，有机半导体（微电子）集成电路工业将形成重要而庞大的电子工业。有机半导体技术可用于有机 EL 显示、柔性显示、纸张显示、I/O 集成接口器件、携带型信息器件、分子集成电路和分子运算单元等。在近期，I/O 集成接口器件将是主要目标。可以说，有机半导体技术将是下一代显示器的技术基础。

1.5.5　数字视频处理

原始数字视频帧数据量大，处理（包括传输、存储等操作）的实时性要求高，这要求系统合理安排帧数据的缓存，设计高性能的视频处理算法，高效地对视频数据进行压缩，以降低对视频数据存储和传输带宽的压力。其中，由于视频数据处理运算量巨大且实时性强，通用处理器难以满足要求，所以往往选用专用 DSP 等微处理器。

数字视频处理的内容非常广泛，包括视频编解码、去噪、增强、运动估计、目标检测、场景分析、视频滤波等。相关的理论和算法将在本书后续章节进行介绍。

1.6　嵌入式数字视频处理系统[34,35,36]

数字视频处理的计算量很大，需要很强的数据处理能力。通用处理器芯片处理这种计算密集的应用时效率低下，如要通过一系列移位和加法运算来实现乘法运算。同时许多视频应用希望能够摆脱对 PC 的依赖，以降低设备成本、提高设备的移动性。嵌入式视频处理系统在这两个方面有明显的优势。

当前有多种嵌入式开发方案可供选择，主要有 ARM、DSP 和 FPGA 等技术。ARM（Advanced RISC Machine）架构是面向低预算市场设计的 RISC 微处理器，它提供一系列内核、体系扩展、微处理器和系统芯片方案，4 个功能模块可根据用户的要求定制。所有产品采用通用的软件体系，所以相同的软件可在所有产品中运行。ARM 可以有效地缩短应用程序开发与测试的时间，也降低了研发费用。

FPGA（Programmable Gate Array，现场可编程门阵列）是专用集成电路（ASIC）中集成度最高的一种。FPGA 采用逻辑单元阵列 LCA 的概念，内部包括可配置逻辑模块、输出输入模块和内部连线三个部分。用户可对 FPGA 内部的逻辑模块和 I/O 模块重新配置，以实现用户的逻辑。它还具有静态可重复编程和动态再系统重构的特性，使得硬件的功能可以像软件一样通过编程来修改。作为 ASIC 领域中的一种半定制电路，FPGA 既解决了定制电路的不足，又克服了原有可编程器件门电路数有限的缺点。FPGA 开发可以大大缩短设计时间，减少 PCB 面积，提高系统的可靠性。FPGA 芯片是小批量系统提高系统集成度、可靠性的最佳选择之一。

DSP 是一种独特的微处理器，有自己的完整指令系统，是以数字信号来处理大量信息的器件。数字信号处理器包括控制单元、运算单元、各种寄存器及一定数量的存储单元等，在其外围还可以连接若干存储器，并可以与一定数量的外部设备互相通信，有软、硬件的全面功能，本身就是一个微型计算机。DSP 采用哈佛设计，程序和数据分别存储在不同的空间，取指令和执行指令可重叠，大大提高了微处理器的速度。DSP 不仅具有可编程性，而且其实时运行速度可达每秒数以千万条复杂指令程序，远远超过通用微处理器，是数字化电子世界中日益重要的计算机芯片。其强大数据处理能力和高运行速度，是最值得称道

的两大特色。DSP 运算能力强、速度快、体积小，而且采用软件编程具有高度的灵活性，因此为从事各种复杂应用提供了一条有效途径。与通用微处理器相比，DSP 的通用功能相对较弱。

那么三者之间的区别是什么呢？ARM 具有比较强的事务管理功能，可用来运行界面及应用程序等，其优势主要体现在控制方面。DSP 主要用来计算，如加密解密、调制解调等，优势是强大的数据处理能力和较高的运行速度。FPGA 可以用 VHDL 等编程，灵活性强，能够进行编程、除错、再编程和重复操作，可以充分地进行设计开发和验证。其现场编程能力可以延长产品在市场上的寿命，这种能力可以用来进行系统升级或除错。

由于视频处理计算量大，同时算法比较复杂且可能需要经常升级，因而基于 DSP 平台的视频处理系统得到了广泛的重视。总的来讲，DSP 平台视频开发有如下优势：

（1）用户开发自由度更大，可迅速满足新要求，增强产品的竞争能力。

（2）DSP 处理能力强，可在一个 DSP 上实现多路音视频信号的处理，同时还提供了视频滤波、解交织、高分辨率显示等视频专用功能，以及网络接口、IDE 接口等通用功能，可以进一步大幅度降低产品成本。

（3）开发周期短，实现快速技术更新和产品换代。

（4）芯片功耗低，对提高产品的稳定性提供可靠保障。

目前，基于 DSP 开发的视频产品非常丰富，如大家熟知的数字电视机顶盒、PDA、视频监控设备、数码摄像机等。这些应用大体上可以分为高性能和便携式两类，因此应针对具体应用的特点选择不同的芯片来实现最佳性能。在高性能应用中，通常数据运算量巨大，因而对芯片主频、片上内存等提出了更高的要求，另外某些产品需要丰富的外围接口，如 IP 视频网络电话等。对便携式视频应用，如数字摄录机，由于都是采用电池驱动，产品设计最大的考虑就是降低功耗，因此必须选择一款低功耗的处理器芯片。所以在对某种应用进行开发之前，必须了解该应用对处理芯片的要求，从而进行成功的器件选型。

市场上当前有多款视频处理的 DSP 主流产品，它们分别具有各自的技术特色。其中，ADI 公司的 Blackfin DSP 是一种很好的选择。美国模拟仪器公司（ADI）是全世界领先的高性能信号处理集成电路制造商，是全球主要的可编程 DSP 芯片供应商之一。Blackfin DSP 是 ADI 与 Intel 联合开发的体现高性能体系结构的首款第四代 DSP 产品，能广泛应用于文本、声音、图像和视频等数字信号的处理。作为一种尖端的数字信号处理基础，Blackfin DSP 的体系结构不仅特别适合于完成视频、音频、语音和数据通信中的数字信号处理，同时还提供综合的控制能力。其主要特点包括微信号结构、动态电源管理、高度并行的计算单元、高性能地址产生器、优秀的代码密度处理能力、专用视频指令、分层内存结构、多种集成式外设、调试和 JTAB 接口等。

Blackfin DSP 还支持一整套软件和硬件开发工具。开发工具为工程师开发和优化 DSP 系统提供了更容易且更具鲁棒性的方法，并缩短产品开发周期，加快上市时间。开发工具主要包括硬件和软件两大部分。硬件部分包括以下 4 类。

（1）EZ-KIT Lite 桌面评估板，包括 VisualDSP++开发环境评估套件。VisualDSP++评估套件仅提供有限的存储器。

（2）EZ-Extender 子板使开发人员能够连接并访问 ADI 公司及第三方合作伙伴的各种外设，从而扩展 EZ-KIT Lite 评估套件的接口。

（3）启动套件提供开始着手多媒体应用开发所需的全部工具。启动套件包括 Blackfin EZ-KIT Lite、EZ-Extenders 子板，以及软件开发套件（SDK）。软件开发套件包含源代码、如何做系列文档及各种编/解码代码，使得开始多媒体应用的开发更容易，并缩短学习曲线。

（4）仿真器（快速片上调试）为开发人员提供了加载代码、设置断点，并观察变量、存储器、寄存器等功能。

软件方面也包括 4 个部分。

（1）VisualDSP++软件开发环境，包括 C++编译器、汇编器、链接器、增强的用户界面、高级绘图工具，以及特性统计图表，使开发人员能够轻松定位编程瓶颈。

（2）软件开发套件（SDK）包含软件范例、源代码、设备驱动程序、算法、实用信息及应用笔记，它们将有助于处理器应用的开发。软件可以与 ADI 处理器相配合使用，作为如何使用特定功能和外设的框架或范例。

（3）软件模块包括音频与视频编解码器、编码器、解码器、后处理代码及其他模块，用来加速处理器的开发与评估。

（4）VisualAudio Designer 与集成了软件开发与调试环境的 VisualDSP++配合工作，本身包括音频系统设计与开发所需的各种已可以使用的软件构建模块，并提供生成可以被产品使用的代码（为存储器与 MIPS 使用而优化）的能力。支持 ADSP-BF533 与 ADSP-BF537 Blackfin 处理器及 ADSP-21262、ADSP-21364 与 ADSP-21369 SHARC 处理器。

Blackfin 处理器系列品种齐全，适合消费、汽车、工业、仪器仪表和通信市场的各类应用。为满足这些市场需求，Blackfin 处理器系列提供可扩展的性能，从 200MHz ADSP-BF535 到双核 600MHz ADSP-BF561 应有尽有。Blackfin 产品还具有极低功耗特性，一些产品的功耗低至 0.23mW/MHz。ADI 公司将信守对 Blackfin 处理器的长期承诺，开发各种单核和双核处理器，不断提升速度、功效并降低成本，确保后续每一代基于 Blackfin 的设计都能从中获益。未来的器件将进一步加强 ADI 公司在性能和功效方面的领导地位，同时保持代码兼容性，提供最广泛的处理器选择，以应对要求最苛刻的融合处理挑战，如图 1.3 所示。

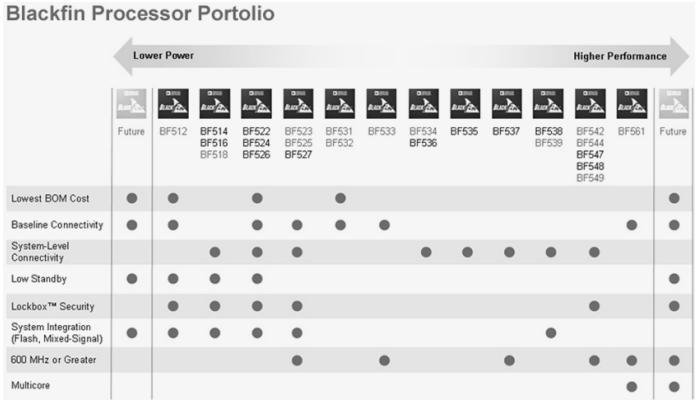


图 1.3　Blackfin 处理器发展规划

1.7 研究现状与发展前景

视频行业的数字化，是模拟世界中数字化较晚的行业之一，主要原因是数字化的视频信息数据量大，应用实现困难。最早实现数字视频的行业是家庭消费的 VCD 和 DVD。20 世纪 90 年代初 VCD 问世，最早实现 VCD 商业化的 C-CUBE 公司，其核心 MPEG1 解码芯片就是基于 DSP 实现的。

当前数字化视频处理技术的核心是数字视频编码/解码技术。在 20 世纪 90 年代初以前，没有人相信视频信息可以数字化处理，但当视频压缩技术取得重大进展时，这种传统的认识被打破。现在，数字视频得到了广泛应用，如 Internet 网上的视频邮件及网络视频游戏；在电信网络上传送视频会议；在计算机网上进行远程视频教学等。这一切得益于视频压缩技术的发展。视频内容的网络化、交互性对视频编解码提出了更高的要求，人们希望进一步提高压缩率、实现基于对象的灵活编码、提高编码内容的交互式操作能力等。

数字视频的发展，是随着数字视频标准的发展而发展的，它将沿着数字化、网络化和智能化的步骤发展。数字化阶段其实就是一个标准化的阶段，现阶段国际两大组织分别制定了 MPEG 标准和 H.26x 的标准，如 MPEG1 带来了 VCD 的兴起，MPEG2 带来了 DVD 的商机，而 H.261、H.263 为可视电话制定标准并为此形成网络化时代的新兴市场。数字视频标准也逐步开始统一，随着 MPEG4 和 H.264 的推出，标准的统一将势在必行。

数字视频监控系统仍是一个市场热点，它已经开始逐步取代传统模拟系统。当前数字监控产品还主要集中在视频信号的数字化、数据压缩、传输、存储、集中管理与远程控制上。各国学者则早已提出了智能化监控的观点，通过对视频中场景和运动的检测与分析，实现自动报警乃至自动处置，既能降低劳动强度，又能够做到实时反应。这里就需要用到运动分析、目标跟踪等视频处理技术，这些理论与技术正在一些高端的监控系统中得到实施，必将称为未来视频监控的方向。

随着视频内容的几何式增长，从资源库中搜索感兴趣的视频内容已经不再那么容易。简单的添加标签的方式对视频内容的检索则不是非常有效，为此基于视频内容的自动检索已经成为一个新的研究热点。其基本思想是通过自动视频分析获取视频特征，如背景亮度、运动强度、有无特定人物出现等，然后就可以根据这些特定信息实现视频内容的有效检索。

随着技术的发展，数字视频的采集、显示、编解码等已经趋于实用，这方面的产品必将逐渐丰富，性能必将不断提高。除此之外，视频处理的智能化也在逐渐提高，能够协助人类处理更为复杂的任务。DSP 平台特别适应于上述数字视频处理需求，因而它必将在各类视频应用中起到重要作用。数字视频处理技术和 DSP 技术的进步必将创造出更多有趣的产品，为人类带来更多更大的便利。让我们充满期待地看着 DSP 在数字视频领域的发展和应用。

第 2 章　数字视频基础

2.1　人类视觉机理

在研究各种成像设备的成像机理与视频处理方法之前，首先来了解一些人类自身的视觉系统。人类视觉系统是人类获取外界图像、视频信息的工具。光辐射刺激人眼时，将会引起复杂的生理和心理变化，这种感觉就是视觉。一方面，视觉是人类最重要、同时也是最完美的感知手段，人类视觉系统机理非常复杂；另一方面，研究人的视觉特性对于图像和视频处理具有重要的指导意义。人类视觉系统的研究包括光学、色度学、视觉生理学、视觉心理学、解剖学、神经科学和认知科学等许多学科领域。下面仅就与视觉处理相关的内容加以简单介绍，作为理解视频处理的基础。

2.1.1　人眼视觉特性[14, 17]

眼球包括眼球壁、眼内腔和内容物、神经、血管等组织，如图 2.1 所示。眼球壁主要分为外、中、内三层。外层由角膜、巩膜组成，角膜是眼球前部的透明部分，含丰富的神经，感觉敏锐，是接受信息的最前哨入口。中层包括虹膜、睫状体和脉络膜三部分。虹膜呈环圆形，位于晶体前，中央有直径为 2.5～4mm 的瞳孔。内层为视网膜，它是一层透明的膜，具有很精细的网络结构及丰富的代谢和生理功能。它是视觉形成的神经信息传递的第一站。视网膜的视轴正对黄斑中心凹。黄斑区是视网膜上视觉最敏锐的区域，其中央为中心凹。黄斑鼻侧约 3mm 处直径为 1.5mm 的淡红色区为视盘，是视网膜上视觉纤维汇集向视觉中枢传递的出眼球部位，无感光细胞，故视野上呈现为固有的暗区，称生理盲点。

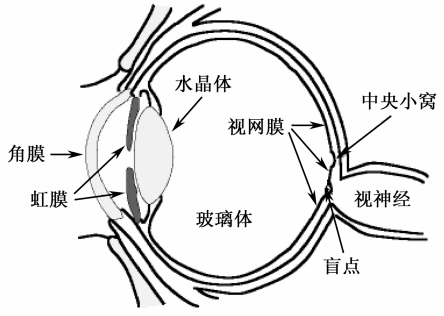


图 2.1　人眼的构造

眼睛之所以能看见物体，光起了很大的作用。太阳光或其他光源照射到物体上，物体吸收了其中一部分色光反射（或投射）的光线进入人类的视觉系统而形成光感。例如，看书的时候，是光先照射到书上，再由书把光反射到你的视网膜上。视网膜上有感光细胞，能把光线的刺激转变成神经信息，通过神经将这些信息传递给大脑，这样就形成了视觉。

视网膜上分布着锥状细胞和柱状细胞两种细胞，前者在强光下工作，能分辨颜色；后者在微光下工作，只能感觉黑白。晚上由于光线非常弱，故而反射光也非常微弱，人眼的视觉又不是十分敏锐，就看不清颜色了。

所以说，人能感觉到图像的颜色和亮度是眼睛的生理结构所决定的。而电影和电视都是根据人眼的视觉特性设计的。电影每秒投射 24 幅静止画面，每幅画面投射两次，由于人眼的视觉惰性，看起来就同活动景象一样。电视每秒扫描 50 幅画面，每幅画面是由 312 根扫描线组成的，由于人眼的视觉惰性和有限的细节分辨能力，看起来就成了整幅的活动景象。人眼的视觉特性是电视技术发展的重要依据。

1．视觉灵敏度

波长不同的可见光光波，给人的颜色感觉不同，亮度感觉也不同，人眼对不同波长光的灵敏度是不同的。人眼的灵敏度因人而异，同一个人眼睛的灵敏度也随年龄和健康状况有所变化，所以采用统计方法，用许多正常视力的观察者来做实验，取其平均值。

经过实验统计分析，国际照明委员会推荐标准视敏度曲线（也称为相对视敏函数曲线）如图 2.2 中的 $V(\lambda)$曲线所示。图中曲线表明具有相等辐射能量、不同波长的光作用于人眼时，引起的亮度感觉是不一样的。可以看出人眼最敏感的光波长为 555nm，颜色是草绿色，这一区域颜色，人眼看起来省力，不易疲劳。在 555nm 两侧，随着波长的增加或减少，亮度感觉逐渐降低。在可见光谱范围之外，辐射能量再大，人眼也是没有亮度感觉的。

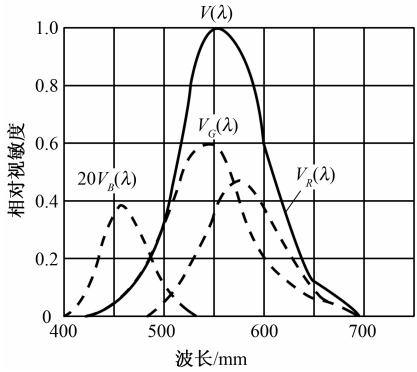


图 2.2　标准视敏度曲线

2．彩色视觉

视网膜上杆状细胞灵敏度很高，但对彩色不敏感，人的夜间视觉主要靠它起作用。因此在暗处只能看到黑白形象而无法辨别颜色。锥状细胞既可辨别光的强弱，又可辨别颜色，

白天视觉主要由它来完成。关于彩色视觉，科学家做过大量实验并提出视觉三色原理的假设，认为锥状细胞又可分成三类，分别称为红敏细胞、绿敏细胞、蓝敏细胞。它们的相对视敏函数曲线分别为图 2.2 所示的 $V_R(\lambda)$、$V_G(\lambda)$和 $V_B(\lambda)$，其峰值分别在 580nm、540nm 和 440nm 处。图中 $V_B(\lambda)$曲线幅度很低，已将其放大了 20 倍。三条曲线的总和等于相对视敏函数曲线 $V(\lambda)$。三条曲线是部分交叉重叠的，很多单色光同时处于两条曲线之下，如 600nm 的单色黄光就处在 $V_R(\lambda)$、$V_G(\lambda)$曲线之下，所以 600nm 的单色黄光既激励了红敏细胞，又激励了绿敏细胞，可引起混合的感觉。当混合红绿光同时作用于视网膜时，分别使红敏细胞、绿敏细胞同时受激励，只要混合光的比例适当，所引起的彩色感觉可以与单色黄光引起的彩色感觉完全相同。

不同波长的光对三种细胞的刺激量不同，产生不同的彩色感觉，因此人眼能分辨各种颜色。电视技术利用这一原理，在图像重现时，不是重现原来景物的光谱分布，而是利用三种相似于红、绿、蓝锥状细胞特性曲线的三种光源进行配色，在色感上得到了相同的效果。

3．分辨力

分辨力是指人眼在观看景物时对细节的分辨能力。对人眼进行分辨力测试的方法如图 2.3 所示。在眼睛的正前方放一块白色的屏幕，屏幕上面有两个相距很近的小黑点；逐渐增加画面与眼睛之间的距离，当增加到一定长度时，人眼就感觉只有一个黑点。这说明眼睛分辨景色细节的能力存在极限值，我们将这种分辨细节的能力称为人眼的分辨力或视觉锐度。

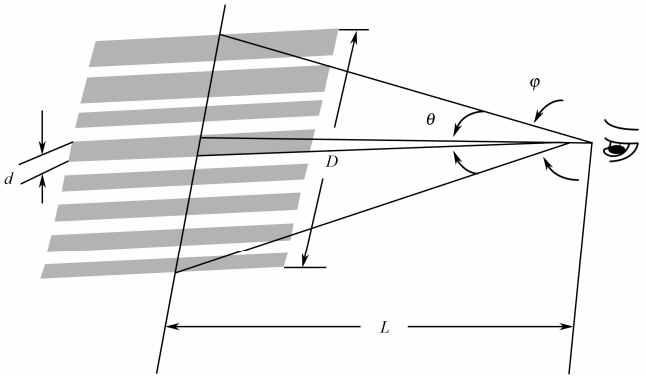


图 2.3　人眼进行分辨力测试的方法

分辨力在很大程度上取决于景物细节的亮度和对比度，当亮度很低时，视力很差，这是因为亮度低时锥状细胞不起作用。但是亮度过大时，视力不再增加，甚至由于眩目现象，视力反而有所降低。此外，细节对比度越小，越不易分辨；观看运动物体时，分辨力更低。

另外，人眼对彩色细节的分辨力比对黑白细节的分辨力低，而且对不同彩色的分辨力也各不相同。实验测得人眼对各种颜色细节的相对分辨力用百分数表示如表 2.1 所示。利用这一特性，在彩色电视系统中传送彩色图像时，只传送黑白图像细节，而不传送彩色细节，这样可减少色信号的带宽。这就是大面积着色原理的依据。

表 2.1　人眼对各种颜色细节的相对分辨力

细 节 颜 色	黑白	黑绿	黑红	黑蓝	红绿	红蓝	绿蓝
相对分辨力（%）	100	94	90	26	40	23	19

4．视觉惰性

当一定强度的光突然作用于视网膜时，不能在瞬间形成稳定的主观亮度感觉，而是按近似指数规律上升；当亮度突然消失后，人眼的亮度感觉并不立即消失，而是按近似指数规律下降。人眼的亮度感觉总是滞后于实际亮度的，这一特性称为视觉惰性或视觉暂留。在中等亮度的光刺激下，视力正常的人视觉暂留时间约为 0.1s。

人眼受到频率较低的周期性的光脉冲刺激时, 会感到一亮一暗的闪烁现象。如果将重复频率提高到某个定值以上，由于视觉惰性，眼睛就感觉不到闪烁了。不引起闪烁感觉的最低重复频率称为临界闪烁频率。临界闪烁频率与光脉冲亮度、亮度变化幅度等相关。人眼临界闪烁频率约为 46Hz。对于重复频率在临界闪烁频率以上的光脉冲，人眼不再感觉到闪烁，这时主观感觉的亮度等于光脉冲亮度的平均值。

除了以上主要特性，人眼还有以下视觉特性：

（1）亮度适应性：人眼由亮环境进入暗环境时开始什么也看不见，经过一段时间适应才能看清物体，称为暗适应，需 30～45min；由暗环境进入亮环境时视觉可以很快回复，称为亮适应性，需 2～3min。

（2）色调对比效应：面积、色度和亮度相同的两个橘红色区域分别处于黄色和红色背景包围下，人眼感觉黄色背景包围的橘红色偏红，红色背景包围的橘红色偏黄。

（3）饱和度对比效应：面积、色度和亮度相同的两个红色区域分别被亮度相同的灰色和红色背景包围，人眼会得到不同饱和度的感觉。

（4）面积对比效应：色度、亮度相同，不同面积的两个彩色区域，面积大的一块会给人以亮度和饱和度都较强的感觉。

（5）马赫效应：人眼对中频成分的响应较高，对高、低频率成分的相应较低。因此在观察亮度跃变时，会感到边缘侧更亮，暗侧更暗。

2.1.2　人类视觉系统模型

人眼类似于一个光学信息处理系统，但由于它具有生物调节的自适应能力，因此它不是一个普通的光学信息处理系统。人眼这种特殊的光学信息处理系统具有非常复杂的特性，根据视觉生理学，研究者们可以建立视觉模型来模拟人类的某些视觉特性。建立视觉模型就是试图用光学系统的概念来模拟某些视觉特性。

1．视觉信息处理模型

从物理结构看，人类视觉系统由光学系统、视网膜和视觉通路组成。图 2.4 给出了人类视觉系统的视觉信息处理模型。它简单模拟了人类视觉系统获取、传输和处理的基本过程。

眼球包括曲光系统和感光系统，曲光系统由角膜、晶体和玻璃体等组成，感光系统即视网膜。视网膜可将输入的光信号转换成生物电信号，电信号通过视神经纤维传递到视神经中枢。由于各个视细胞产生的电信号不同，大脑就形成了景象的感觉。

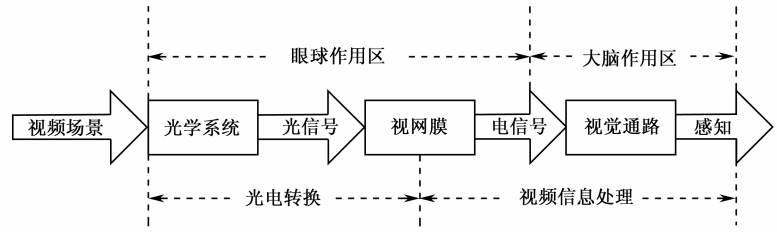


图 2.4　视觉信息处理模型

2．黑白视觉模型

图 2.5 给出了黑白视觉模型。低通滤波器模拟人眼的光学系统，高通滤波器反映了侧抑制引起的马赫效应，对数运算器反映了视觉的亮度恒定现象。所谓亮度恒定现象是指当对景物的亮度、对比度保持一致时，即使景物和背景的亮度在很大的范围内变化，人眼对景物的亮度感觉仍然保持不变。

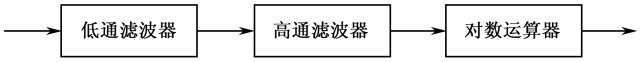


图 2.5　黑白视觉模型

3．彩色视觉模型[13, 14]

图 2.6 是 O.D.F 于 1979 年提出的一个彩色视觉模型，其中 $I(x,y,\lambda)$ 为彩色图像，$S_R(\lambda)$、$S_G(\lambda)$、$S_B(\lambda)$ 为三个彩色滤波器，$R(x,y)$、$G(x,y)$、$B(x,y)$ 为彩色滤波器输出的三种颜色。该模型的第一级反映了人类视觉的三基色理论，对应的公式为式 2-1。

$$\begin{cases} R(x,y)=\int_{\lambda} I(x,y,\lambda)S_R(\lambda)\mathrm{d}\lambda \\ G(x,y)=\int_{\lambda} I(x,y,\lambda)S_G(\lambda)\mathrm{d}\lambda \\ B(x,y)=\int_{\lambda} I(x,y,\lambda)S_B(\lambda)\mathrm{d}\lambda \end{cases} \tag{2-1}$$

第二级反映了视细胞对光强的非线性响应，即

$$\begin{cases} R^*(x,y)=\lg R(x,y) \\ G^*(x,y)=\lg G(x,y) \\ B^*(x,y)=\lg B(x,y) \end{cases} \tag{2-2}$$

三对互相对立的颜色对分别为红—绿、黄—蓝和黑—白对，反映了在视觉通路上的响应，L 为亮度输出，C_1、C_2 为彩色输出，即

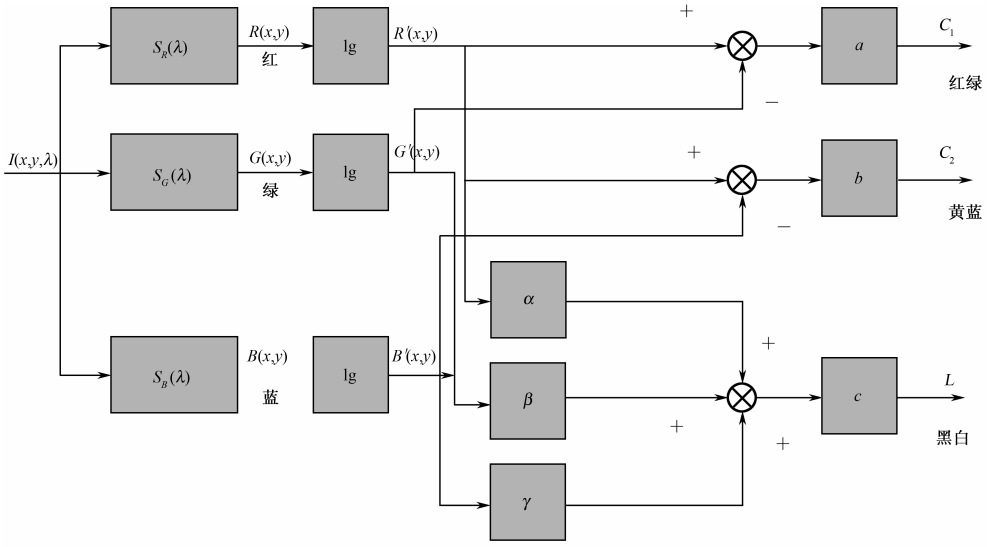


图 2.6　彩色视觉模型

$$\begin{cases} C_1 = a\left[R^*(x,y) - G^*(x,y)\right] = a\lg\dfrac{R(x,y)}{G(x,y)} \\ C_2 = b\left[R^*(x,y) - B^*(x,y)\right] = b\lg\dfrac{R(x,y)}{B(x,y)} \\ L = c\left[\alpha R^*(x,y) + \beta G^*(x,y) + \gamma B^*(x,y)\right] \\ \qquad = c\left[\alpha\lg R(x,y) + \beta\lg G(x,y) + \gamma\lg B(x,y)\right] \end{cases} \tag{2-3}$$

式中，a、b、c、β、γ 为常数。

2.2　颜色感知与表示模型[51]

2.2.1　颜色感知机理

彩色刺激与彩色感觉不是一种简单的因果关系，人眼彩色感觉取决于可见光谱中的不同成分和光照环境。若光的强度相同而波长不同，则引起的视觉效果也不同。改变光的波长，不仅颜色感觉不同，而且亮度也不相同。例如，人眼感到最暗的是暗红色，其次是蓝色和紫色，而最亮的是黄绿色。

在自然界中，光的颜色与波长是一一对应的。而物体的颜色通常是指在日光下物体所呈现的颜色，它与物体对光的反射特性、透射特性有关。表示人眼视觉对颜色感觉的参量有亮度（Luminance）、色调（Hue）和饱和度（Saturation）。

亮度是指人眼对光的明亮程度的感觉，光源的亮度正比于光通量，而物体的亮度不仅取决于物体反射（或透射）光的能力，也取决于照射该物体的光源的辐射功率。反射（或

透射）光的能力越强，即反射（或透射）系数越大，物体就越明亮；照射物体的辐射功率越大，物体就越明亮。通常彩色光越强则感觉就越明亮；反之则越暗。

色调表示颜色的类别，如红色、蓝色、绿色。彩色物体的色调取决于物体在光照下所反射的光谱成分，不同波长的反射光使物体呈现不同的色调。对于某些透光的物体（如玻璃灯），其色调取决于透射光的波长。彩色物体的色调既取决于物体的吸收特性和反射或透射特性，也与照明光源的光谱成分有关。

饱和度是指彩色光所呈现彩色的深浅程度。通常同一色调的彩色光，其饱和度越高，颜色就越深，如深绿色等，反之颜色就越浅。

色度是指色调和饱和度的合称，它既反映了彩色光的颜色，又反映了颜色的深浅程度。用亮度、色调和饱和度这三个参量就能描述出彩色光。

非彩色光由于没有色度，故只有亮度描述。例如，白、灰、黑，它们之间只有亮度的差别，灰色处于黑色与白色之间。

2.2.2　颜色模型

各种颜色可以用一个三维空间来描述，称为彩色空间。为了定量描述和处理的需要，建立合适的颜色模型非常重要。颜色模型中每个空间点都代表某一特定的彩色。不同颜色模型面向不同的应用场合，具有不同的特性，大体上可以分为面向硬设备和面向视觉感知两大类。常用的彩色模型包括 RGB、XYZ、HSI、YUV 等。下面做简单介绍。

1．RGB 颜色空间

计算机彩色显示器与彩色电视机都是采用 R、G、B 相加混色的原理，通过发射出三种不同强度的电子束，使屏幕内侧覆盖的红、绿、蓝磷光材料发光而产生颜色的。这种颜色的表示方法称为 RGB 颜色空间表示。在多媒体计算机技术中，用得最多的是 RGB 颜色空间表示。根据三基色原理，用基色光单位来表示光的量，则在 RGB 颜色空间，任意色光 F 都可以用 R、G、B 三色不同分量的相加混合而成。

$$F = r[R] + g[G] + b[B] \tag{2-4}$$

RGB 颜色空间可以用一个三维的立方体来描述，如图 2.7 所示。自然界中任何一种色光都可由 R、G、B 三基色按不同的比例相加混合而成，当三基色分量都为 0 时混合为黑色光；当三基色分量都为 1（最强）时混合为白色光。任一颜色 F 都是这个立方体坐标中的一点，调整三色系数 r、g、b 中的任一系数都会改变 F 的坐标值，也即改变了 F 的颜色值。RGB 颜色空间采用物理三基色表示，因而物理意义很清楚，适合彩色显像管工作。然而这一体制并不适应人的视觉特点。因而，产生了其他不同的颜色空间表示法。

2．HSI 颜色空间

HSI 颜色空间是从人的视觉系统出发，用色调、色饱和度和亮度来描述颜色。HSI 颜色空间可以用一个圆锥空间模型来描述，如图 2.8 所示。

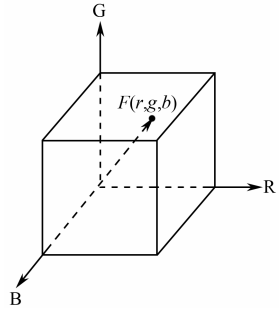


图 2.7　RGB 颜色空间表示

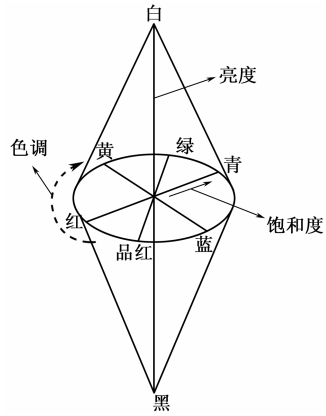


图 2.8　HIS 颜色空间模型

HSI 颜色空间的圆锥模型相当复杂，但却能把色调、亮度和色饱和度的变化情形表现得很清楚。图 2.8 中是 HIS 颜色空间的线条示意图，描述了亮度、色度和饱和度的关系，纵轴表示亮度，亮度值是沿着圆锥的轴线度量的，按照不同的灰度等级，最亮点为纯白色、最暗点为纯黑色，圆锥纵切面描述了同一色调的不同亮度和饱和度的关系；圆锥横切面中色调 H 为绕着圆锥截面度量的色环，圆周上的颜色为完全饱和的纯色；色饱和度为穿过中心的半径横轴。

为了便于颜色处理和识别，人类视觉系统经常采用 HSI 模型，它比 RGB 更符合人类视觉特性。图像处理和视频处理的大量算法采用 HSI 模型会更加方便，因为它将亮度和色度独立开来，大大简化了图像分析和处理的工作量。

HSI 颜色空间和 RGB 颜色空间只是同一物理量的不同表示法，因而它们之间存在着转换关系，如公式所示

$$\begin{cases} I = \dfrac{R+B+G}{3} \\ H = \dfrac{1}{360}\left[90 - \mathrm{Arctan}\left(\dfrac{F}{\sqrt{3}}\right) + \{0, G > B; 180, G < B\}\right] \\ S = 1 - \left[\dfrac{\min(R,G,B)}{I}\right] \end{cases} \tag{2-5}$$

其中

$$F = \frac{2R - B - G}{G - B} \tag{2-6}$$

3．YUV 颜色空间

现代彩色电视系统中常用的是 YUV 颜色空间，其亮度信号 Y 和色度信号 U、V 是分离的。彩色电视采用 YUV 空间正是为了用亮度信号 Y 解决彩色电视机与黑白电视机的兼容问题，使黑白电视机也能接收彩色信号。根据美国国家电视制式委员会 NTSC 制式的标准，当白光的亮度用 Y 来表示时，它和红、绿、蓝三色光的关系可用如下的方程描述

$$Y = 0.3R + 0.59G + 0.11B \tag{2-7}$$

这就是常用的亮度公式。YUV 颜色空间与 RGB 颜色空间的转换关系如下

$$\begin{bmatrix} Y \\ U \\ V \end{bmatrix} = \begin{bmatrix} 0.300 & 0.590 & 0.11 \\ -0.15 & -0.29 & 0.44 \\ 0.61 & -0.52 & -0.096 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \tag{2-8}$$

如果要由 YUV 空间转化成 RGB 空间，只要进行相应的逆运算即可。

4．YIQ 模型

YIQ 模型与 YUV 模型非常类似，是彩色电视制式中使用的另一种重要的颜色模型。这里 Y 表示亮度，I、Q 是两个彩色分量。YIQ 和 RGB 的对应关系用下面的方程式表示

$$\begin{aligned} Y &= 0.299R + 0.587G + 0.114B \\ I &= 0.596R - 0.275G - 0.321B \\ Q &= 0.212R - 0.523G + 0.311B \end{aligned} \tag{2-9}$$

5．YCrCb 模型

YCrCb 模型适用于计算机显示器，它也是使用 Y、Cr 和 Cb 来分别表示一种亮度分量信号和两种色度分量信号。YCrCb 模型与 RGB 空间的转换关系如下

$$\begin{aligned} Y &= 0.299R + 0.578G + 0.114B \\ Cr &= (0.500R - 0.4187G - 0.0813B) + 128 \\ Cb &= (-0.1687R - 0.3313G + 0.500B) + 128 \end{aligned} \tag{2-10}$$

2.3　视频获取与显示

视频获取的方法大体有以下 4 种：

（1）计算机产生彩色图形、静态图像和动态图像；

（2）用彩色扫描仪，扫描输入彩色图形和图像；

（3）用视频信号数字化仪输入到计算机中，可获得静态或动态图像；

（4）用数码相机或数码摄像机直接获取。

2.3.1　彩色视频成像原理[13]

简言之，视频记录了从一个观测系统（人眼或摄像机）所观测的场景中的物体发射或反射光的强度 $C(X,t,\lambda)$。一般地说，该强度在时间和空间上都有变化。令 $a_c(\lambda)$ 表示摄像机的光谱吸收函数，那么摄像机可见的三维空间中的光强度分布为

$$\overline{\psi}(X,t) = \int_0^{\infty} C(X,t,\lambda) a_c(\lambda) \mathrm{d}\lambda \tag{2-11}$$

摄像机在任何时刻 t 摄取的图像函数都是三维光分布在二维图像平面上的投影。令 $P(\cdot)$ 表示投影算子，则三维点 X 的二维位置由 $x=P(X)$ 给出，而 $X=P^{-1}(x)$ 就是二维点 x 对应的三维位置。这样投影图像与三维图像的关系为

$$\psi(P(X),t)=\overline{\psi}(X,t) \quad 或 \quad \psi(x,t)=\overline{\psi}(P^{-1}(X),t) \tag{2-12}$$

函数 $\psi(x,t)$ 称为视频信号，它描述了时刻 t 投影到图像平面 x 的三维位置 X 的辐射强度。通常视频信号具有有限的时间和空间范围，分别对应摄像机的观测区域和摄取持续时间。图像平面中的一点称为像素。对于大多数摄像系统，$P(\cdot)$ 可近似为一个透视投影。

为了感知所有的可见图像，根据三基色的视觉原理，需要三个频率相应类似于所选基色的彩色匹配函数的传感器，多数彩色摄像机采用红色、绿色和蓝色传感器来获得彩色。

2.3.2 视频摄像机

当前有两种基本类型的摄像机，一种是基于光电管的摄像机，如光导摄像管、氧化铅摄像管或正析摄像管；另一种是固态传感器，如电荷耦合器件（CCD）。基于光电管的摄像机用电子束一行一行地扫描（称为光栅扫描），每一帧中的扫描行被转换成用不同电压代表不同光强的电子信号。不同的扫描行是以相继的方式在略微不同的时间上摄取的，逐行扫描是相继地扫描每一行，隔行扫描则是在半帧的时间隔一行扫描一行。CCD 摄像机的光敏表面由二维传感器矩阵组成，每个传感器对应一个像素，到达每个传感器的光信号被转换成一个电信号。在每帧时间内摄取的传感器值首先存储在传感器中，然后一次次地相继输出以形成光栅信号。因此同一帧中记录的值是同时被摄取的。

为了摄取彩色，通常有三种类型的光敏表面或 CCD 传感器，每个传感器的频率响应决定于所选基色的彩色匹配函数。为减少成本，多数消费类摄像机采用单片 CCD 进行彩色成像，即把每个像素的传感器区域分成 3 个或 4 个子区域，每个子区域感应不同的基色。摄取的三个彩色信号可以转换成一个亮度信号和两个色度信号，并作为分量彩色图像发送出去，或者复用一个复合信号。现在多数摄像机是基于 CCD 的，因为同样空间分辨率下它可以做得更小更轻，能够以很小体积的芯片摄取很高分辨率的图像矩阵。

2.3.3 视频显示

视频显示最普通的设备是阴极摄像管（CRT）。在 CRT 监视器中，电子枪一行一行地向屏幕发射电子束以激励荧光粉，其强度正比于在相应位置的视频信号的强度。为显示彩色图像，三个电子枪发射三个电子束，在每个位置以期望的强度组合激励红色、绿色和蓝色荧光粉。为了更精确，每个彩色像素由排列成小三角形的三个元素组成，称为三元组。CRT 具有很大的动态范围，显示的图像可以很亮，足以在白天或者很远处观看。然而，为使电子到达屏幕边界，CRT 宽度需要大致与屏幕宽度相当，因而不适用于在小型轻便的设备中。

为克服这个问题，人们开发了各种不同的平板显示器。一种流行的设备是液晶显示器（LCD）。LCD 的主要思想是通过施加电场改变光学特性进而改变液晶的亮度和彩色。电场

由一个晶体管阵列产生或刷新。例如，在 LCD 中采用驱动矩阵薄膜晶体管（TFT）或采用等离子。等离子技术消除了对薄膜晶体管的需求，从而使大屏幕 LCD 成为可能。

刚刚介绍的光栅扫描和显示机制只用于视频摄像机和显示器。运动图像摄像机在任何帧瞬间摄取的彩色图像全部记录在胶片上，然后再用模拟光学投影系统回放相继记录的帧。

2.3.4　复合视频与分量视频[51]

理想情况下，彩色视频应该由三个函数规定，每个函数描述一个彩色分量，这种格式的视频称为分量视频。由于历史的原因，也存在各种复合视频格式，其中的三个彩色信号被复用成一个单独的信号。当彩色电视系统首次开发出来后发明了这种复合格式，当时要求彩色电视信号的传输方式能使黑白电视机从中抽取亮度分量。构造复合函数依赖于这样一个性质，即色度信号拥有比亮度分量小得多的带宽。通过将每个色度分量调制到一个位于亮度分量高端的频率上，并把已调色度分量加到原始亮度信号，就产生了一个包含亮度和色度信息的复合信号。为了在彩色监视器上显示复合视频信号，用滤波器把已调色度信号从亮度信号中分离出来，然后把产生的亮度和色度信号转换成红色、绿色和蓝色分量。对于灰度级显示器，只提取亮度信号并直接显示。

现在所有模拟电视系统都以复合格式传输彩色电视信号。复合格式也用于将视频存储在某种模拟磁带上。除了与灰度级信号兼容以外，复合格式消除了处理彩色信号时使不同彩色分量同步的需要。复合信号的带宽比三个分量信号带宽的总和小，因此能被更有效地传输或存储。然而这些优点是以牺牲图像质量为代价的：经常由于色度和亮度的串扰而形成可察觉的人工痕迹。图 2.9 显示了黑白全电视信号的组成。彩色全电视信号的场、行同步信号与之兼容，另外还在行消隐脉冲中增加了 9～11 个副载波周期，如图 2.10 所示。

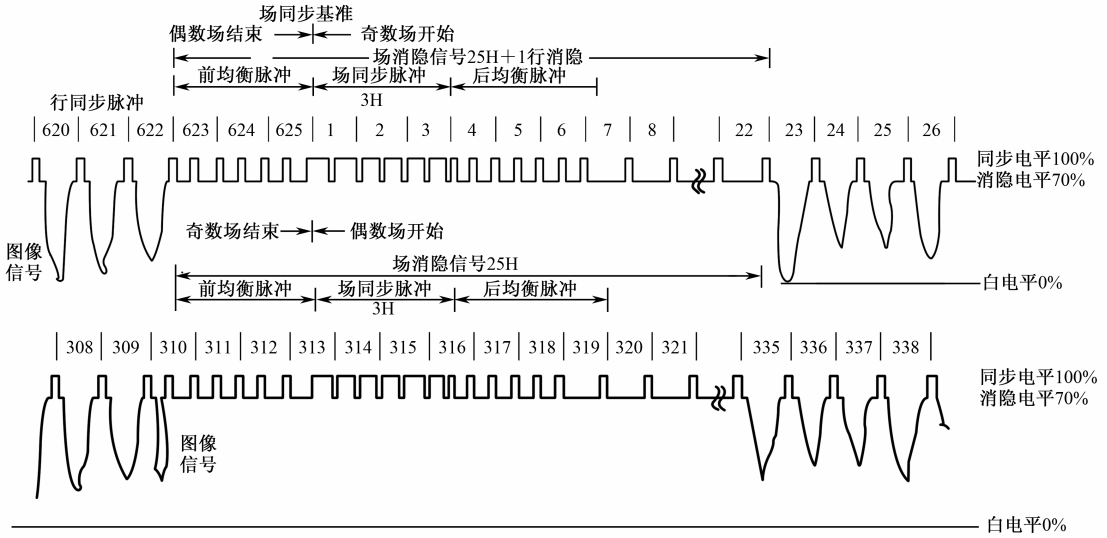


图 2.9　黑白全电视信号的组成

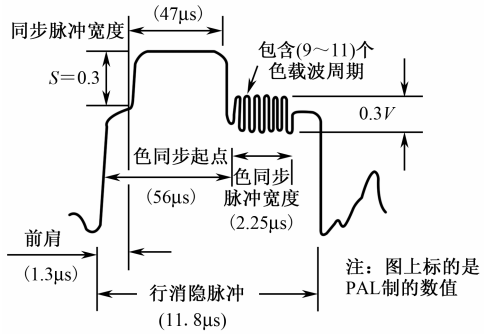


图 2.10　彩色电视系统的水平消隐间隔

2.3.5　伽马校正

由摄像机摄取的视频帧反映了成像景物的彩色值。事实上，多数摄像机输出的信号并不是与实际的彩色值呈线性关系，而是一种非线性的形式，即

$$v_c = B_c^{\gamma_c} \tag{2-13}$$

这里 B_c 表示实际的光亮度，v_c 是摄像机的输出电压。γ_c 的取值范围从 1.0（对于多数 CCD 摄像机）到 1.7（对于光导管摄像机）。类似地，大多数显示设备的输入电压值与显示的彩色强度之间也有这种非线性的关系，即

$$B_d = v_d^{\gamma_d} \tag{2-14}$$

对于 CRT 显示器，典型的 γ_d 是 2.2～2.5。为呈现真实的彩色，必须在摄像机的输出端加入一个相反的幂函数。类似地，在发送要显示的真实图像值之前，必须对显示设备的“伽马效应”进行预补偿。这些过程被称为伽马校正。

理想情况下的电视广播，在发送端，被电视摄像机摄取的 RGB 值应该首先基于摄像机的伽马值进行校正，然后转换到用于传输的彩色坐标；在接收端，收到的 YIQ 或 YUV 值应该首先被转换成 RGB 值，然后用监视器的伽马值补偿。为了在数以百万计的接收机中减少该处理过程，广播视频信号在 RGB 域进行预伽马校正。令 v_c 表示摄像机摄取的 RGB 信号，则用于显示的已经过伽马校正的信号由下式得到

$$v_d = v_c^{\gamma_c/\gamma_d} \tag{2-15}$$

在多数电视系统中，采用比率 γ_c/γ_d=2.2。经过伽马校正的值转换成 YIQ 或 YUV 进行传输，接收机则将其转换为 RGB 进行显示。

2.4　模拟视频技术[13]

2.4.1　模拟视频信号

模拟视频信号已经应用了几十年，至今仍然在使用。最常见的通用视频标准包括 NTSC

和 PAL 制式，现代消费模拟传输系统包括了 S-Video、分量视频、专业 GBR 视频及计算机 RGB 系统。所谓模拟视频就是采用电子学的方法来传送和显示活动景物或静止图像。它通过在电磁信号上建立变化来支持图像信息的传播和显示，大多数家用电视机和录像机显示的都是模拟视频。模拟电影，如静态透明胶片或者通常的运动图像电影，是通过在一条醋酸盐胶片上建立颜料的色彩变化来支持图像信息的传送和播映，大多数的模拟电影在邻近该帧的轨道上还支持光学声音信息。在 CRT 显示器上看到的图像是电视显像管内发出的一系列连续的线，线数的多少决定了视频的质量，NTSC 为 525 线、PAL 为 625 线。在模拟视频世界中，视频表现为一系列连续波动的信号。

2.4.2　视频光栅扫描

模拟电视系统通常用光栅扫描方式，即在一定的时间间隔内电子束以从左到右、从上到下的方式扫描荧光屏表面。若时间间隔为一帧图像的时间，则获得或显示的是一帧图像；若时间间隔为一场图像的时间，则获得或显示一场图像。在电视系统中，两场图像为一帧。扫描方式通常有逐行扫描和隔行扫描。

逐行扫描如图 2.11 所示，其中实线为行扫描正程，电子束从左到右扫过的轨迹；虚线是行扫描逆程，电子束从右到左扫过的轨迹。行扫描周期为 $T_h = T_{ht} + T_{hr}$，其中，T_{ht} 为电子束在屏幕上从左到右扫完一行正程所需的时间；T_{hr} 为从右返回到左所需的时间。逐行扫描可以减少屏幕大面积闪烁和边缘闪烁，分解力高，图像更清晰稳定。

图 2.11　逐行扫描示意图

隔行扫描如图 2.12 所示，一幅图像由奇数场（场正程）和偶数场（场逆程）组成。场扫描周期为 $T_v = T_{vt} + T_{vr}$，其中，T_{vt} 为场扫描正程的时间；T_{vr} 为场扫描逆程的时间。采用隔行扫描的目的是为了压缩光电转换后产生的视频信号的频带。例如，在电视中采用一帧图像隔行后分两场顺序传输，可以压缩一半频带而不明显降低图像质量。但这是以牺牲系统性能为代价的。例如，行间闪烁效应会使人眼产生视觉疲劳，垂直分解力下降导致图像质量下降等。

光栅扫描的基本参数是帧频 f_p（每秒取样帧数）和每帧扫描行数 n，它们分别对应于光栅扫描的时间分辨率和空间垂直方向分辨率，若图像的高度为 H，则有：$f_p = 1/\Delta t$，$n = H/\Delta y$，其中，Δy 为行间隔（或垂直取样间隔），Δt 为帧间隔时间。视频就是由许多间隔为 Δt 的帧组成的。根据人眼视频的惰性，过低的帧频会使人眼感觉到图像的跳跃和闪

烁，而每帧扫描行数越多，图像就越清晰。电影的帧频为 24 帧/s；电视系统通常采用 25～30 帧/s 的隔行扫描方式，而计算机显示器的帧频通常采用大于 72 帧/s。

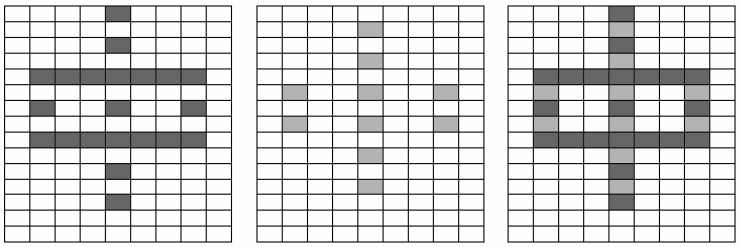


图 2.12　隔行扫描示意图

2.4.3　模拟电视系统

在模拟电视信号中，模拟视频信号用于传送画面，模拟音频信号用于传送声音。模拟电视系统标准主要有 NTSC 制、PAL 制和 SECAM 制。NTSC 制主要用于北美和日本，PAL 制主要用于欧洲和中国，SECAM 制主要用于前苏联和东欧国家。从技术上讲，NTSC 制的频带最经济，而 PAL 制和 SECAM 制对传输线路的要求较低。表 2.2 给出了三种模拟电视系统标准的主要技术指标。

表 2.2　模拟电视系统标准的主要技术指标

相 关 参 数	PAL	NTSC	SECAM
帧频（帧/s）	25	29.97	25
场频（场/s）	50	59.94	50
每帧行数（行/帧）	625	525	625
有效行数（行/帧）	576	480	576
行频（行/s）	15625	15750	15625
行周期（μs）	64	63.5	64
有效扫描周期（μs）	52	53.5	52
行消隐时间（μs）	12	10	12
场周期（ms）	20	16.7	20
场消隐时间（μs）	1612	1333	1568
图像宽高比	4∶3	4∶3	4∶3
彩色分量	YUV	YIQ	YUV
亮度带宽（MHz）	5.0，5.5	4.2	6.0
色度带宽（MHz）	1.3（U,V）	1.6(I),0.6(Q)	1.0(U,V)
复合信号带宽（MHz）	8.0	6.0	8.0
彩色副载波（MHz）	4.43	3.58	4.25(Db)，4.41（Dr）
音频副载波（MHz）	5.5,6.0	4.5	6.5
色度调制方式	QAM	QAM	FM

通常彩色视频获取和显示都采用 RGB 三基色方式，但 RGB 信号不能直接用在黑白电视机上；同时 RGB 视频信号的带宽很大，对传输或存储容量的要求很高。因此，在彩色电视系统中采用复合视频信号来传输。PAL 制彩色电视复合信号的带宽为 8MHz，NTSC 制为 6MHz，亮度信号和色度信号在该带宽内传输。

PAL 制复合信号经调制后的两个色度信号 U（t）和 V（t）分别为

$$\begin{cases} u(t) = U(t)\sin(w_{sc}t) \\ v(t) = V(t)P(t)\cos(\omega_{sc}t) \end{cases} \tag{2-16}$$

式中，$\omega_{sc} = 2\pi f_{sc}$ 为色度信号的副载波角频率，$P(t)$为开关信号。正交幅度调制（QAM）色度信号为

$$c(t) = u(t) + v(t) = C(t)\sin[\omega_{sc}t + \theta(t)] \tag{2-17}$$

式中

$$\theta(t) = P(t)\arctan[V(t)/U(t)]$$
$$C(t) = \sqrt{U^2(t) + V^2(t)}$$

若 $P(t) = +1$（偶数场）或 $P(t) = -1$（奇数场），则可表示彩色副载波逐行倒相的 PAL 制色度信号。

在 PAL 制中，为了简化滤波器的设计，亮度信号采用残留单边带调制（VSB），残留带宽为 1.2MHz。色度信号的副载波频率 f_{sc} =4.43MHz，由于色度信号副载波处亮度分量的能量很低，因此色度信号的副载波放在亮度信号频谱的高频段，即在亮度信号的高频段插入经过正交幅度调制的两个色度分量，从而形成彩色电视的复合视频信号

$$e(t) = Y(t) + c(t) = Y(t) + C(t)\sin[\omega_{sc}t + \theta(t)] \tag{2-18}$$

式中，当 $C(t) = 0$ 时就是黑白电视信号，它可以看做是彩色电视信号的特殊情况。

NTSC 制复合视频信号采用 YIQ 彩色模型，亮度信号采用残留单边带调制，亮度信号采用正交平衡调幅。两个色度信号 $I(t)$和 $Q(t)$经过调制后分别为

$$\begin{cases} i(t) = I(t)\sin(\omega_{sc}t) \\ q(t) = Q(t)\cos(\omega_{sc}t) \end{cases} \tag{2-19}$$

式中，$\omega_{sc} = 2\pi f_{sc}$ 为色度信号的副载波角频率。色度信号为

$$c(t) = i(t) + q(t) = C(t)\sin[\omega_{sc}t + \theta(t)] \tag{2-20}$$

式中，$\theta(t) = \arctan[I(t)/Q(t)]$，$C(t) = \sqrt{I^2(t) + Q^2(t)}$。

2.5　数字视频技术[15,16,21]

2.5.1　模拟视频信号数字化表示

数字视频是指用二进制数字表示的视频信号，数字视频既可直接来源于数字摄像机（如 CCD 摄像机等），也可以将模拟视频信号经过数字处理变成数字视频信号。

彩色视频数字化分为复合编码和分量编码。复合编码是直接对彩色全电视信号进行 PCM 编码，分量编码是对亮度分量 Y 和色差分量 θ_Y 、C_R（或三基色 R、G、B）分别进行 PCM 编码。分量编码图像质量高，且信号便于三种电视制式之间进行节目交换，因此，分量编码得到了广泛的应用。目前几乎所有的数字视频系统均采用分量编码。RGB 到 YC_BC_R 的转换关系为

$$\begin{pmatrix} Y \\ C_B-128 \\ C_R-128 \end{pmatrix}=\begin{pmatrix} \frac{77}{256} & \frac{150}{256} & \frac{29}{256} \\ -\frac{44}{256} & -\frac{87}{256} & \frac{231}{256} \\ \frac{131}{256} & -\frac{110}{256} & -\frac{21}{256} \end{pmatrix}\begin{pmatrix} R \\ G \\ B \end{pmatrix} \tag{2-21}$$

YC_BC_R 到 RGB 的转换关系为

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix}=\begin{pmatrix} 1.0000 & 1.4020 & 0.0000 \\ 1.0000 & 0.0000 & 1.7720 \\ 1.0000 & -0.7140 & -0.3441 \end{pmatrix}\begin{pmatrix} Y \\ C_B-128 \\ C_R-128 \end{pmatrix} \tag{2-22}$$

模拟视频信号经过数字化处理后，就变成一帧帧数字图像组成的图像序列，即数字视频信号。每帧图像由 N 行、每行 M 个像素组成，共有 $M\times N$ 个像素。利用人眼的视觉惰性，每秒连续播放 30 帧（帧频 f_p）以上，就能给人以较好的连续运动景物的感觉。每个像素用 N_b 比特表示，数字视频信号的信息传输速率为 $M\times N\times f_p\times N_b$。

例如，PAL 制彩色数字视频信号用 RGB 分量（N_b=3×8=24bit）表示时，帧频 f_p=25（每秒 25 帧），M=576（每帧 576 行），N=720（每行 720 像素点），其信息传输速率为 720×576×25×24≈249Mbps。

2.5.2 数字视频的特点及应用

由于模拟视频的特性，它在传输、存储和交互等方面具有很大的局限性。例如，在普通模拟电视中，只有频道选择等简单功能；在盒式磁带录像机（VCR）中，只能进行快速搜索和慢速重放等操作；模拟视频的录制、存储非常不方便，且多次录制、存储时噪声积累严重；传输时所叠加的噪声（即使很小）很难消除和分开；对信道的线性特性要求较高，放大器的非线性会产生波形畸形；随着传输距离的增加，噪声积累越来越大，使模拟视频信号的传输质量恶化；微分增益、微分相位失真会带来彩色失真，等等。

与模拟视频相比，数字视频具有很多优点：便于传输和交换，便于多媒体通信，便于存储、处理和加密，无噪声积累，差错可控制，可通过压缩编码来减低数码率，便于设备的小型化，信噪比高，稳定可靠，交互能力强等。

随着数字电路和微电子技术的进步，特别是超大规模集成电路的快速发展，使得数字视频的优点变得越来越突出，应用越来越广泛，如高清晰度电视（HDTV）、多媒体、视频会议、移动视频、监视控制、医疗设备、航空航天、军事、教育、电影等。目前，数字视

频用于桌面和掌上技术已经成熟，已成为消费电子产业的重要支柱，如数字电视、数码照相机和数码摄像机等。数字视频将会给计算机、通信和电子消费等产业带来革命性的变化。

2.5.3　ITU-T BT.601 数字视频标准

为了便于国际节目交换及 625 行 PAL 制系统与 525 行 NTSC 制系统之间的兼容，1982 年 CCIR（国际无线电通信咨询委员会）制定了 CCIR601 数字视频标准，1993 年变更为国际电信联盟无线电通信部门 ITU-T BT.601 建议，定义了对应于 525 行和 625 行电视演播室的数字编码参数，如表 2.3 所示。三个分量信号 Y、R-Y、B-Y 的抽样频率分别为 13.5MHz、6.75MHz 和 6.75MHz，三个分量在一行中抽样点数的比例为 4∶2∶2，且每帧的行数相同，故简称为 4∶2∶2 标准。

4∶2∶2 标准是为演播室制定的对图像质量要求较高的分量编码标准。对图像质量要求更高的应用场合，可采用 4∶4∶4 标准；对图像质量要求较低的应用场合，如信号源本身的分辨率就低（如家用录像机、新闻采访摄像机信号等），可采用 4∶1∶1 标准或 4∶2∶0 标准。

表 2.3　ITU-T BT.601 建议的 4:2:2 标准

参　　量		NTSC 制（525 行，60 场）	PLA 制（525 行，60 场）
编码信号		Y/R-Y/B-Y	
全行采样点数	亮度 Y	858	864
	色度 R-Y/B-Y	429	432
采样结构		正交，按行/场/帧重复，每行中的 R-Y/B-Y 取样与奇数（1、3、5、…）点 Y 取样同位	
采样频率（MHz）	亮度 Y	13.5	
	色度 R-Y/B-Y	6.75	
编码方式	亮度信号和色差信号均为 PCM 8 比特		
每行有效采样点数	亮度 Y	720	
	色度 R-Y/B-Y	360	
有效图像尺寸	亮度 Y	720×480	720×576
	色度 R-Y/B-Y	360×480	360×576

2.6　视频模型[13,14]

视频记录了在摄像机所观测的场景中物体运动的状态及其变化方式。视频信号是从动态的三维景物投影到视频摄像机图像平面上的一个二维的图像序列，即视频是一个三维信号，具有两个空间维度（构成图像平面）和一个时间维度。

由于真实世界景物的运动状态及其变化方式非常复杂，为了有效地描述真实世界的变化，需要建立一系列视频模型，如照明模型、摄像机模型、物体模型和场景模型等。场景模型用于描述包括照明光源、物体和摄像机的世界，即描述运动物体与一个三维场景的摄

像机是如何互相定位的。在视频编码中，通常假定物体与摄像机的成像平面平行运动，即使用二维场景模型。三维场景模型能更有效的描述真实世界。根据不同的照明模型、摄像机模型和物体模型可得到不同的场景模型。下面介绍照明模型、摄像机模型和物体模型。

2.6.1 照明模型

照明模型主要用于描述照明变化引起的视频信号在时间上的变化。照明模型可分为光谱模型和几何模型。光谱模型适用于多种彩色光源（或由于不同彩色物体反射的间接光源），几何模型适用于环境光源（照射物体时不会产生阴影）和点光源（如聚光灯）。对每种类型的光源，又可以分为局部照明模型和总体照明模型。局部照明模型假定照明模型与物体的位置无关，总体照明模型要考虑物体间的影响（如阴影等）。

光源有两种：照明光源和反射光源。照明光源的色彩感觉取决于光的波长范围。照明光源遵循相加规则。反射光源指能反射入射光的光源，通常的表面既有漫反射也有镜面反射，但只有漫反射才能显示物体表面的颜色。反射光辐射强度的分布与入射光的光强 $f_i(L,V,\overline{N},P,t,\lambda)$ 和物体表面的反射系数 $r(L,V,\overline{N},P,t,\lambda)$ 有关，即

$$f_r(L,B,\overline{N},P,t,\lambda)=r(L,V,\overline{N},P,t,\lambda)\cdot f_i(L,B,\overline{N},P,t,\lambda) \tag{2-23}$$

式中，P 为物体表面的位置，L 为照明方向，V 为 P 点与摄像机焦点的观测方向，$\overline{N}$ 为点 P 处的表面法线矢量，λ 为光的波长。反射系数 $r(L,V,\overline{N},P,t,\lambda)$ 为反射光的强度与入射光的强度之比。例如，假定照明方向 L 和观测方向 V 固定不变，则式（2-23）可简化为

$$f_r(\overline{N},P,t,\lambda)=r(\overline{N},P,t,\lambda)\cdot f_i(\overline{N},P,t,\lambda) \tag{2-24}$$

当只有环境光源 $f_a(t,\lambda)$，且物体表面为漫反射时，其反射光强度的分布为

$$f_r(P,t,\lambda)=r(P,t,\lambda)\cdot f_a(t,\lambda) \tag{2-25}$$

当只有点光源时，对于局部照明模型和漫反射表面，物体表面上任意点 P 处的反射光强度取决于入射光方向 L 与该点处的表面法线 N 之间的夹角 θ，即

$$f_r(P,t,\lambda)=r(P,t,\lambda)\cdot f_p(t,\lambda)\cdot\cos\theta \tag{2-26}$$

式中，$f_p(t,\lambda)$ 为点光源的最大光强，即光垂直于表面时的光强。

当多个环境光源和点光源都存在时，任意一点反射光强度的分布是该点对每个光源反射光强的叠加。

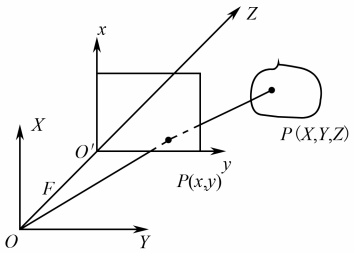
2.6.2 摄像机模型

摄像机模型描述真实场景中物体在摄像机成像图像平面上的投影，即实现四维空间 (X,Y,Z,t) 到三维空间 (x,y,t) 的映射。这里，(X,Y,Z) 为三维空间坐标系（也称为世界坐标系），(x,y) 为二维投影图像平面。

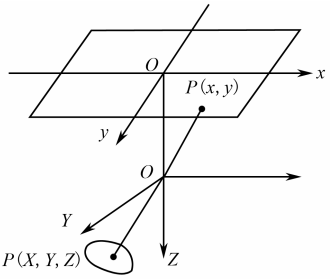
1．透视投影

透视投影也称为中心投影。以摄像机为中心，观察空间中的物体，可以获得物体在二

维图像平面上的投影图像，如图 2.13 所示。其中，原点 O 为透视中心，$OO'=F$ 为焦距，表示观察者与投影图像平面之间的距离。从 O 观测空间中物体上一点 $P(X,Y,Z)$，得到图像平面上投影点 $p(x,y)$。O、$P(X,Y,Z)$ 和 $p(x,y)$ 三点在一条直线上。满足 $\frac{x}{F}=\frac{X}{Z}$，$\frac{y}{F}=\frac{Y}{Z}$ 的结构称为透视投影（或中心投影），即以观察者为中心的投影模型。



（a）摄像机在观测物体和成像图像平面的一侧



（b）摄像机在观测物体和成像图像平面之间

图 2.13　透视投影

2．正交投影

当物体距摄像机很远时，中心投影可用正交投影（即平行投影）来近似（如图 2.14 所示），有

$$x=X,\quad y=Y$$

或

$$\begin{pmatrix}x\\y\end{pmatrix}=\begin{pmatrix}1&0&0\\0&1&0\end{pmatrix}\begin{pmatrix}X\\Y\\Z\end{pmatrix}$$

这里 x,y 为投影图像的平面坐标。

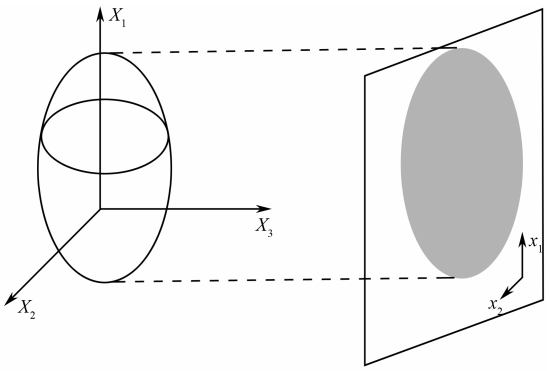


图 2.14　正交投影

3．摄像机运动

摄像机的典型运动形式有：跟（track，摄像机沿成像图像平面的水平轴平移）、吊（boom，

摄像机沿成像图像平面的垂直轴平移）、推（dolly，摄像机沿光轴方向的平移）、摇（pan，摄像机绕垂直轴旋转）、倾（tilt，摄像机绕水平轴旋转）、滚（roll，摄像机绕光轴旋转）和变焦（摄像机改变其焦距）。

跟和吊时摄像机沿 $OXYZ$ 坐标系 X 和 Y 轴平移。设摄像机实际平移为 T_x 和 T_y，则有

$$\begin{pmatrix} X' \\ Y' \\ Z' \end{pmatrix} = \begin{pmatrix} X \\ Y \\ Z \end{pmatrix} + \begin{pmatrix} T_x \\ T_y \\ 0 \end{pmatrix} \tag{2-27}$$

利用 $x = F\dfrac{X}{Z}$，$y = F\dfrac{Y}{Z}$，摄像机在成像图像平面 $xO'y$ 的二维空间位置变化为

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} \dfrac{FT_x}{Z} \\ \dfrac{FT_y}{Z} \end{pmatrix} \tag{2-28}$$

显然，成像图像平面中心 (x, y) 的平移取决于它所对应的三维空间点的 Z 坐标。

摇和倾时摄像机绕 $OXYZ$ 坐标系 X 和 Y 轴旋转。设摄像机旋转角为 θ_x 和 θ_y，则有

$$\begin{pmatrix} X' \\ Y' \\ Z' \end{pmatrix} = \overrightarrow{\boldsymbol{R}_x}\overrightarrow{\boldsymbol{R}_y} \begin{pmatrix} X \\ Y \\ Z \end{pmatrix} \tag{2-29}$$

其中，$\overrightarrow{\boldsymbol{R}_x}$ 和 $\overrightarrow{\boldsymbol{R}_y}$ 分别为摄像机绕 X 轴和 Y 轴的旋转矩阵，即

$$\overrightarrow{\boldsymbol{R}_x} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\theta x & -\sin\theta x \\ 0 & \sin\theta x & \cos\theta x \end{pmatrix} \tag{2-30}$$

$$\overrightarrow{\boldsymbol{R}_y} = \begin{pmatrix} \cos\theta y & 0 & \sin\theta y \\ 0 & 1 & 0 \\ -\sin\theta y & 0 & \cos\theta y \end{pmatrix} \tag{2-31}$$

当旋转角 θ_x 和 θ_y 均很小时，有

$$\overrightarrow{\boldsymbol{R}_x}\overrightarrow{\boldsymbol{R}_y} = \begin{pmatrix} 1 & 0 & \theta_y \\ 0 & 1 & \theta_{-x} \\ -\theta_y & \theta_x & 1 \end{pmatrix} \tag{2-32}$$

若 $Y\theta_x << Z$，$X\theta_y << Z$，则 $Z' = Z$。利用 $x = F\dfrac{X}{Z}$，$y = F\dfrac{Y}{Z}$，可得到摄像机在成像图像平面 $xO'y$ 的二维空间位置变化，即

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} F\theta_y \\ -F\theta_x \end{pmatrix} \tag{2-33}$$

变焦时，设 F 和 F' 分别为摄像机变焦前后的焦距，由 $x = F\dfrac{X}{Z}$，$y = F\dfrac{Y}{Z}$ 可得

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \mu x \\ \mu y \end{pmatrix} \tag{2-34}$$

其中，$\mu = \dfrac{F'}{F}$ 称为变焦系数。其二维运动场为

$$\begin{pmatrix} \mathrm{d}_x(x,y) \\ \mathrm{d}_y(x,y) \end{pmatrix} = \begin{pmatrix} (1-\mu)x \\ (1-\mu)y \end{pmatrix} \tag{2-35}$$

滚是指摄像机绕 Z 轴旋转，即

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos\theta_z & -\sin\theta_z \\ \sin\theta_z & \cos\theta_z \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \tag{2-36}$$

摄像机绕 Z 轴的旋转角 θ_z 很小时，有

$$\begin{pmatrix} x' \\ y' \end{pmatrix} \approx \begin{pmatrix} 1 & -\theta_z \\ \theta_z & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \tag{2-37}$$

2.6.3　物体模型

物体模型是关于真实物体的假设。所谓物体是指在一个场景中可以分离的实体，如一辆车、一台电视、一个人等。一个物体可用形状、运动和纹理等特征来描述。其中，纹理模型可用于描述一个物体表面的特性。下面简要介绍形状模型和运动模型。

1．形状模型

一个三维物体的形状由它所占的三维空间来描述。通常由于人们不太关注物体的内部，因此可用物体的表面来描述它的形状。一般可采用三角形网络（即线框）的方法，三角形网格是用位于物体表面的控制点（顶点）来构建的。控制点的数量和位置取决于物体的形状和三角形网络模型对物体形状描述的精度。

2．刚体运动模型

当控制点不能被独立地移动和不能改变物体的形状时，该物体就是刚性的；否则，就是柔性的。一个物体可以是刚性的或柔性的。刚性物体在三维空间中的运动可以分解为围绕通过原点的一个轴的旋转和平移。在三维空间中，物体的旋转可用一个 3×3 的矩阵来描述，平移可用一个 3×1 的列向量来描述。

假设在三维空间中一个运动物体上的特征点为 p，运动前（在 t_1 时刻）的坐标为 $p(x,y,z)$；运动后（在 t_2 时刻）相应的点 p' 的坐标为 $p'(x',y',z')$，运动前后应满足

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \bar{\boldsymbol{R}} \begin{pmatrix} x \\ y \\ z \end{pmatrix} + \bar{\boldsymbol{T}} \tag{2-38}$$

式中，$\vec{\boldsymbol{R}}$ 为旋转矩阵，定义为 $\vec{\boldsymbol{R}}=\begin{pmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{pmatrix}$；$\vec{\boldsymbol{T}}$ 为平移向量，定义为 $\vec{\boldsymbol{T}}=\begin{pmatrix} \Delta x \\ \Delta y \\ \Delta z \end{pmatrix}$，$\Delta x$，$\Delta y$，$\Delta z$ 分别为运动物体在 x,y,z 三个方向上的平移量。

例如，若一个物体绕三维空间的原点转动，则它的旋转矩阵为

$$\begin{aligned} \vec{\boldsymbol{R}} &= \vec{\boldsymbol{R}_x}\vec{\boldsymbol{R}_y}\vec{\boldsymbol{R}_z} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\theta_x & -\sin\theta_x \\ 0 & \sin\theta_x & \cos\theta_x \end{pmatrix}\begin{pmatrix} \cos\theta_y & 0 & \sin\theta_y \\ 0 & 1 & 0 \\ -\sin\theta_y & 0 & \cos\theta_y \end{pmatrix}\begin{pmatrix} \cos\theta_z & -\sin\theta_z & 0 \\ \sin\theta_z & \cos\theta_z & 0 \\ 0 & 0 & 1 \end{pmatrix} \\ &= \begin{pmatrix} \cos\theta_y\cos\theta_z & \sin\theta_x\sin\theta_y\cos\theta_z-\cos\theta_x\sin\theta_z & \cos\theta_x\sin\theta_y\cos\theta_z+\sin\theta_x\sin\theta_z \\ \cos\theta_y\sin\theta_z & \sin\theta_x\sin\theta_y\sin\theta_z+\cos\theta_x\cos\theta_z & \cos\theta_x\sin\theta_y\sin\theta_z-\sin\theta_x\cos\theta_z \\ -\sin\theta_y & \sin\theta_x\cos\theta_y & \cos\theta_x\cos\theta_y \end{pmatrix} \end{aligned} \tag{2-39}$$

显然，该矩阵 $\vec{\boldsymbol{R}}$ 是一个正交矩阵。

数字视频处理相关的内容还有很多，比如上/下行变换、超分辨率、视频水印、视频三维重建等。具体内容请参考相关的参考文献[13～17]。

第 3 章　数字信号处理与嵌入式开发

3.1　数字信号处理基础及 DSP 系统应用[4~9,28,37]

信号在生活中无处不在，我们经常遇到的信号有语音、音乐、图片和视频信号等。信号是自变量（如时间、位置、温度、压力和距离等）的函数。例如，语音信号表示空间上某个点的空气压力，它是时间的函数。电视中的视频信号是由一帧帧的图像序列组成的，是一个三个变量的函数：两个空间坐标和一个时间坐标。

信号携带着信息，信号处理的目的就是提取信号携带的有用信息。信息提取的方法取决于信号的类型及信号中信息的性质。同样的信号在不同的表达空间中有不同的表现形式（如时域表达和频域表达），适用于各种不同的应用。对于模拟信号，大多数信号处理运算都是在时域进行的。对于离散信号，时域和频域运算都广泛使用。时域中的基本运算是标乘、延时、相加、积分和微分等。滤波是使用最广泛的信号处理运算，它的目的是根据要求改变频谱。实现滤波运算的系统称为滤波器，如低通滤波器可以滤除高频噪声，得到更为平滑的信号，同时降低信号带宽；带通滤波器则可以方便地提取出对应频带范围内的子信号，是实现频分复用的基础。

数字信号处理技术的发源可以追溯到 17 世纪，当时开发了有限差分方法、数值积分方法和数字内插方法，用以解决涉及连续变量和函数的物理问题。20 世纪 50 年代，随着数字计算机的出现，数字信号处理开始兴起。最初的应用主要涉及模拟信号处理方法的仿真。大约在 20 世纪 60 年代初，数字信号处理本身逐渐被作为一个独立的领域，从此数字信号处理理论和应用得到了巨大的发展和突破。模拟信号的数字化处理包括三个基本步骤：模数转换、数字信号的处理、数模转换。

数字信号处理过程中可以很容易地调整处理器特性，如可以很方便地实现自适应滤波器。数字处理可以实现用模拟系统不可能实现的某些特性，如精确的线性相位及多抽样率处理。与模拟电路不同，数字电路可以在没有负载匹配问题的情况下进行级联。数字信号的存储、传输几乎不会丢失信息，也适合于进行离线处理。另外，数字处理可以用于频率非常低的信号，而模拟处理所需要的电感和电容在物理上必将具有很大的尺寸。

数字信号处理也有一些不足。一个明显的缺点是对模拟信号进行数字处理将增加系统的复杂性。这是由于需要附加一些预处理和后处理设备，如 ADC 和 DAC 及相关的滤波器和其他复杂的数字电路。此外，数字信号有效处理的频率范围有限，这主要是由抽样和保持电路及 ADC 决定，它将受到技术和工艺发展的限制。另外，数字系统是由耗电的有源器件构成的，而大量模拟算法的实现可以用无源电路。而且通过 ADC 获得的大量数字信号的

存储和处理，对数字处理部分的处理速度和内存消耗提出了很高的要求。

综合分析，在很多应用中，数字信号处理系统的优点远远超过其缺点，而且数字电路工艺的迅速发展使得硬件成本不断下降、性能不断上升，因此数字信号处理的应用正在迅速增加，已经占据了信号处理的主流地位。

数字信号处理的基本算法有两种：滤波算法和信号分析算法。这些算法的表示基于递归或非递归的差分方程，或者离散傅里叶变换，而且这些算法的实现可以采用硬件、软件或软硬件结合的方式。无论以哪种方式处理，都希望数字信号的处理过程尽可能快，以满足自然世界中实时性的要求。

绪论部分介绍了数字信号处理器（DSP）的基本概念和知识，现在我们开始介绍其在嵌入式数字信号处理中的应用。图 3.1 显式了如何在 MP3 音频播放器中使用 DSP。在录音阶段，模拟音频信号通过接收机或其他信号源进行输入。然后该模拟信号通过模数转换器转换为数字信号并传至 DSP 处理器。DSP 处理器对数字信号进行 MP3 编码，并将压缩后的文件保存到存储器中。在回放阶段，DSP 处理器将文件从存储器中取出，然后对其进行解码，最后通过数模转换器将信号转换为模拟信号，通过扬声器系统输出。在更复杂的音频应用系统中，DSP 将执行其他功能，如音量控制、自动增益均衡、自动回声消除等。

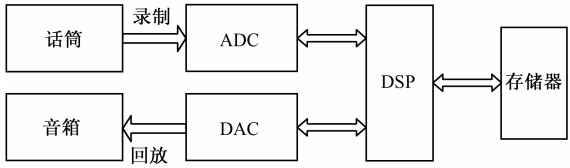


图 3.1　DSP 在 MP3 音频播放器中的应用

虽然上述过程也能用模拟方式现实，但用数字方式处理信号具有高速、高精度、可编程等优点。为了理解模拟电路与 DSP 系统的差别，可以通过比较两个系统实现滤波器功能来分析。常见的模拟滤波器使用电阻、电容、电感、放大器等来实现。模拟电路价格便宜、易于组装，但是难以校准、修改和维护，难度随滤波器阶数呈指数增长。基于 DSP 实现的滤波器则可以更容易地设计和修改，因为 DSP 实现的滤波器函数是基于软件实现的，灵活且可准确重复。另外，为创建高阶响应的、灵活可调的滤波器，只需要软件修改，而无需另外的硬件，这与纯模拟电路是不同的。

如当逼近一个理想滤波器时，利用纯模拟电路实现如此精度的逼近将必然是高成本的。同时高阶纯模拟滤波器的响应随着时间的变化则不够稳定。而基于 DSP 的滤波器只需要具有合适的参数及足够高的阶数即可。在 DSP 系统中，编程实现高阶 FIR 滤波器不是一个困难的任务。另外，至关重要的一点是利用 DSP 逼近理想滤波器是长期稳定的，可编程的 DSP 会一次次地得到同样的响应。

如图 3.2 所示，DSP 系统的设计主要解决以下几个问题：真实世界信号的采集与数字化、数字信号处理算法、对外部信号作出响应等。下面分别讨论。

（1）采样真实世界信号。首先利用某种转换器将声、光、热、电、磁等物理现象的连续能量级变化转化成可测量的电压或电流信号，然后利用 ADC 将信号数字化。ADC 的采

样频率至关重要，它是由所需的信号最小信息量决定的。根据奈奎斯特采样定律，采样率必须是模拟信号最高频率的两倍以上。对于信号中超过最高频率的元素（如噪声），必须首先利用低通滤波器滤除。该滤波器称为抗混叠滤波器，具有一个有限频率的滚降，所以必须提供额外的带宽作为该滤波器的过渡带。例如，20kHz 的输入信号带宽可能需要 2～4kHz 的额外带宽。

数据获取既可以一次采样一个样本（基于样本的处理系统），也可以一次采样一帧样本（基于帧的处理系统）。实时系统不能丢失任何数据，所以 DSP 处理当前帧的同时还必须获取下一帧数据。在获取数据方面，DSP 有特殊的结构特性来实现：利用处理器的灵活数据寻址加上 DMA 通道功能来实现无缝的数据获取。

（2）数字信号处理的实现。DSP 需要在收到下一个样本前完成当前样本相关的所有操作，样本间隔就是 DSP 完成所有处理任务允许的时间，如 48kHz 的音频采样率就对应于 20.833μs 的采样间隔。给定采样率，处理速度要受到算法复杂度的影响，复杂算法需要更多地处理任务。因为样本间隔是固定的，复杂度越高，处理能力就要求越高。假设算法需要 50 步运算，样本间隔为 20.833μs，则所需最小的 DSP 处理器速度为 50/20.833μs=2.4Mbps。

（3）响应真实世界的信号。DSP 系统必须预留时间来响应外部设备、控制输入/输出数据的流动等，如来自 ADC 的中断信号。DSP 对中断的响应时间直接影响其真正用于处理信号的时间。中断反应时间依赖于几个因素，最重要的是 DSP 结构的指令流水线，更多的流水线级别会导致更长的中断响应时间。具有 DMA 和双内存空间结构特征的 DSP 可以很容易避免该浪费，能够让 DSP 接收和存储数据，而不中断处理器。

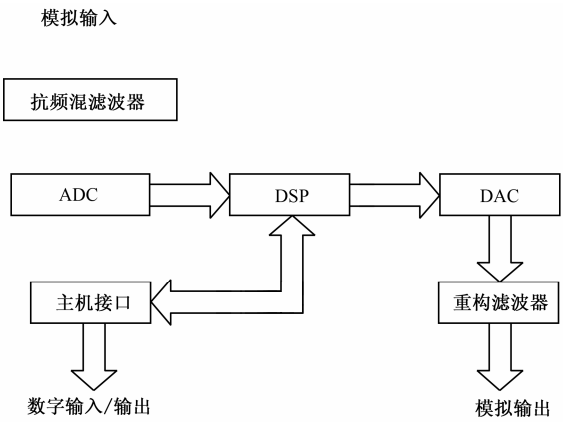


图 3.2　DSP 系统构成模块图

由于处理系统的大部分功能来自于可编程的 DSP，所以构成 DSP 系统的部件很少，这就为 DSP 系统的设计和开发带了很大的便利。如图 3.2 所示，DSP 系统中数据流的要点如下。

（1）模拟输入。模拟信号利用抗混叠滤波器进行带限滤波，然后施加到 ADC 输入端。在选定的采样时刻，转换器中断 DSP 处理器以使该样本成为可用的。ADC 与 DSP 之间采取串行接口还是并行接口依赖于数据量和设计复杂度的折中，此外还要考虑空间、功耗和

价格等。

（2）数字信号处理。输入数据由 DSP 算法软件处理。处理器完成所需的计算时，将结果送给 DAC。因为信号处理是可编程的，处理数据是很灵活的，可以通过改进程序提高性能。

（3）模拟输出。DAC 在下一个采样时钟将 DSP 输出转化为期望的模拟输出。转换器的输出利用低通重建滤波器来产生重建的模拟信号。

（4）主机接口。可选的主机接口使 DSP 可以与外部系统通信，发送并接收数据和控制信息。

在硬件上，DSP 主要包含以下重要部件。

（1）程序存储器：存储 DSP 用于处理数据的程序。

（2）数据存储器：存储即将被处理的信息。

（3）计算引擎：执行数学和逻辑处理，访问程序存储器中的程序及数据存储器中的数据。

（4）输入/输出：提供连接外部世界的各种功能。

下面以 ADI Blackfin 系列处理器为例详细介绍其系统构成与特性，然后介绍基于 DSP 系统的开发过程，以及开发工具 VisualDSP++的基本使用。

3.2 Blackfin 处理器简介[28,37~39]

美国模拟器件公司（ADI）是全球领先的高性能信号处理器集成电路制造商，是全球主要的可编程 DSP 芯片供应商之一。在通用 DSP 市场上，ADI 公司的占有率名列前茅。Blackfin DSP 是 ADI 与 Intel 联合开发的体现高性能体系结构的首款第四代 DSP 产品。目前，公布的 Blackfin 系列 DSP 有 BF51x, BF52x, BF531/532/533，BF535，BF537, BF538/538F, BF539, BF54x, BF561 等。

传统上，嵌入式微控制器工程师和数字信号处理工程师通过不同的手段独立完成设计。现在，通过外部事件和应用之间的复杂互动，控制和信号处理从根本上交织在一起。由于处理性能不断增加，可编程处理器已经成为很多高性能信号处理系统中的关键技术，经常用在 ADI 高性能模拟产品的相同应用场合和信号链中。汇聚能力和不断增长的处理器性能的结合为 ADI 公司处理器产品家族带来了新的机会。Blackfin 处理器包含了一种新的 16/32 位嵌入式处理器，非常适合汇聚能力起关键作用的应用——多格式音频、视频、语言和图像处理；多模式基带和分组处理；控制处理和实时安全性。这种软件灵活性和可扩展性的独特结合为 Blackfin 处理器赢得了汇聚应用领域广泛的适应性，如数字家庭娱乐、网络和流媒体、汽车远程信息处理和信息娱乐、数字无线电及移动电视。

Blackfin 处理器突出特性包括以下方面：

（1）单指令集结构达到或超过竞争 DSP 产品范围的处理性能。

（2）提供更低的功耗、成本和更高的存储效率。

（3）适合下一代嵌入式应用的 16/32 位结构。

（4）单核完成控制、信号和多媒体处理。

（5）通过动态电源管理进行信号处理或电源消耗的性能调节。

（6）代码和引脚兼容的产品线。

（7）性价比高，扩展了其在多种终端产品中的工程开发。

（8）具有竞争 DSP 双倍性能和一半的功耗，允许指标突破和新的应用。

（9）通过多个工具链和操作系统的支持，可在千百种设计里迅速采用。

（10）提升了开发者的生产力。

（11）强大的软件开发环境和内核性能满足最小优化需求。

（12）广泛的第三方合作伙伴减轻了产品开发的风险。

（13）具有业界领先的开发工具、实时操作系统、软件提供商和系统集成伙伴的支持。

下面介绍其中几款典型的 Blackfin 处理器的特性及其主要应用。

（1）ADSP-BF527。ADSP-BF527 是一款外设先进的低功率 Blackfin 处理器，其功能模块框图如图 3.3 所示。它具有对代码和内容的硬件安全保护，内核主频高达 600MHz（1200MMACS），双通道、全双工的同步串口支持 8 路立体声 I^2S 通道，12 个外设 DMA 通道支持一维和二维数据传输，具有 10/100 以太网接口，内存控制器提供到多个 SDRAM、SRAM、Flash 或 ROM 内存组的无缝连接，支持并行外设接口 PPI。它主要的应用场合包括 VoIP（IP 电话）、多媒体应用处理器、网络音频、工业控制、测量仪器和成像系统等。

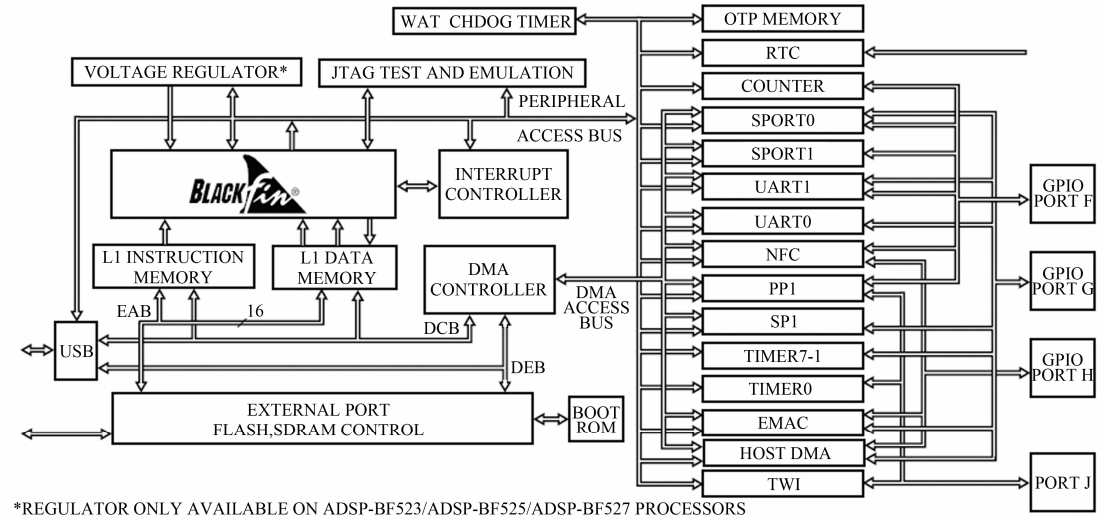


图 3.3　ADSP-BF522～527 处理器功能模块框图

BF527 外设支持到网络设备（如以太网或 WiFi 802.11 a/b/g 模块）的无缝连接，以及功率利用率高的信号处理，因而非常适合有 IP 连接的应用，如 VoIP 电话或 VoIP 摄像机。VoIP 对嵌入式设计者的挑战是选择一个成本高效、易于发布、性能可伸缩的处理方案。一个比较理想的嵌入式解决方案是设计一个能实现低通道的基本 VoIP 方案的平台，同时保留足够

的性能来开发增值性能和服务，如视频、音乐、成像和系统控制。与传统的 VoIP 嵌入式方案不同（它们使用两个处理器核来提供 VoIP 功能），ADSP-BF527 使用统一的内核结构来提供汇聚的解决方案，以允许声音和视频信号处理与 RISC MCU 同时工作，处理网络和用户接口需求。由于能够在一个汇聚处理器上提供全 VoIP 功能，这为开发者提供了统一的软件开发环境、更快的系统调试和发布及更低的整体系统成本。

知识产权保护已经成为当今嵌入式计算应用的必要部分，ADSP-BF527 提供了一套安全策略，很好地平衡了灵活性、可升级性与性能之间的关系。其做法是引入基于固件的解决方案，其中包含了一次性可编程（OTP）内存，使得用户可以实现安全访问程序代码的私有密钥。

（2）ADSP-BF533。ADSP-BF533 是一款高性能的通用 Blackfin 处理器，其功能模块框图如图 3.4 所示。它为当今的大多数高要求的汇聚式信号处理应用提供了高性能、高效功率的处理器选择。它具有高性能的 16/32 位 Blackfin 嵌入处理器内核、灵活的 Cache 结构、增强的 DMA 子系统及动态功率管理（DPM）概念，为系统设计者提供了一个灵活的平台，可以解决包括消费产品、通信、汽车、仪器和工业应用等各种应用方案。它主频最高达 600MHz，具有 UART 和定时器，还有内核电压调整器。它的外设接口具有无缝的用于数据获取的通用转换器的连接，包括视频捕获和显示接口，同时具有大量片上 SRAM 来最大化系统性能。BF533 主要用于多媒体应用、家庭音视频、嵌入式 MODEM、测量仪器、成像、工业控制、音频通信等场合。

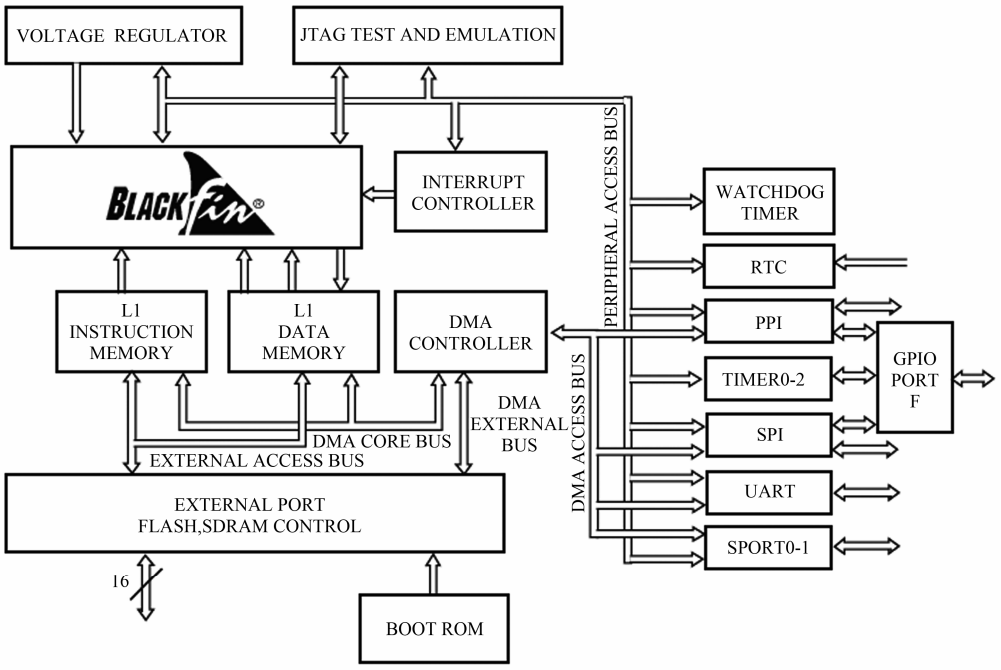


图 3.4　ADSP-BF531/532/533 功能模块框图

（3）ADSP-BF548。ADSP-BF548 是一款高性能汇聚多媒体 Blackfin 处理器，其功能模块框图如图 3.5 所示，内核最高频率为 533MHz（1066 MMACS）。ADSP-BF548 处理器主要针对对性能和成本有高要求的汇聚性多媒体应用，如先进汽车信息系统、移动通信、安全和访问控制、工业与仪表。它集成了多媒体、人机界面和丰富的外设，具有更大的系统带宽和片上内存，为客户提供了设计大多数高要求应用的平台。对于需要额外外部内存的应用，ADSP-BF548 族产品提供了各种变形，能够与标准 DDR1 或 1.8V 低功耗 DDR 内存设备连接。ADSP-BF54x 同样提供了与 BF527 类似的知识产权保护策略。

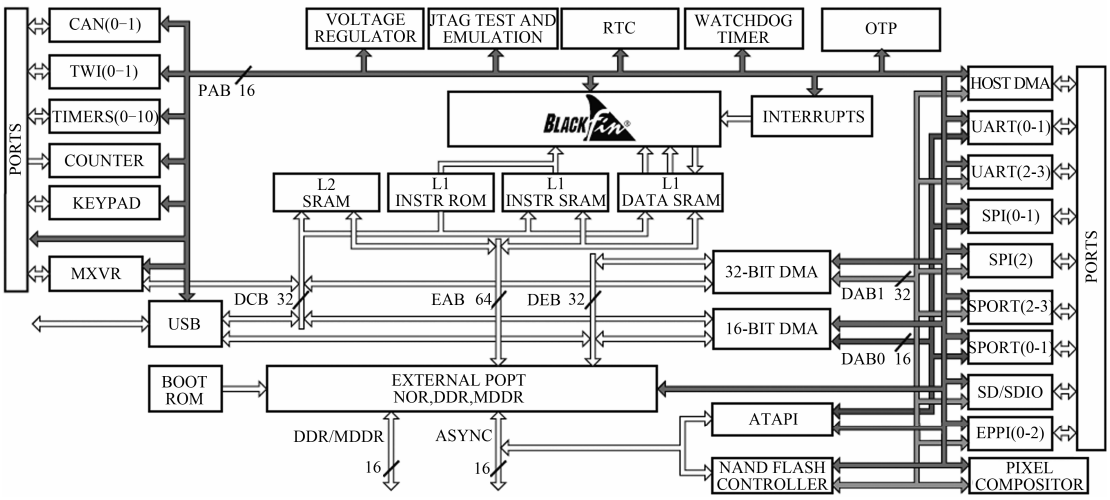


图 3.5　ADSP-BF542/544/547/548/549 功能模块框图

ADSP-BF548 提供了外设灵活性来补充其高性能处理。这些丰富的系统级外设很适合先进汽车信息系统和工业多媒体应用，可以处理多个流行的标准和不同的性能要求。为了增强连接性，BF548 集成了高速 USB OTG 模块，它支持 8/16 位主机 DMA 接口。这些系统级外设，以及标准串行连接（多个片上 SPORT、SPI、UART、TWI 和 CAN 接口），为多个片外设备提供了无缝连接，包括消费类和通信类产品、蓝牙设备等。这个级别的集成对新兴的且不断变化的产品和标准特别有效，如汽车信息系统和多媒体产品等。

为了连接片外存储媒体，包括硬盘、CD/DVD 驱动器和 NAND Flash 产品，ADSP-BF548 实现了 SD/SDIO 控制器、ATAPI-6 控制器和 8/16 位 NAND Flash 控制器。ADSP-BF548 还提供了一些多媒体增强，通过硬件集成降低处理器的 MIPS、扩展显示功能、缩短客户开发时间。支持 ITU-R BT.656 视频格式的多个增强 PPI 接口能够驱动 18/24 位 LCD 显示。硬件加速模块、像素合成器可以执行叠加、颜色转换和 Alpha 混合。该模块显著降低了处理器内核在软件 RGB-YUV 颜色转换和 Alpha 混合上的开销。

对于人机界面能力，ADSP-BF548 提供了 32 位上/下计数器，能够感知来自工业驱动器或人工拇指轮的两位面积或二进制代码。此外还有一个 8×8 键盘接口。

（4）ADSP-BF561。ADSP-BF561 是一款对称多处理器消费类多媒体产品，它扩展了 Blackfin 处理器系列处理器的性能界限，具有两个高性能 Blackfin 处理器内核、灵活的 Cache

体系结构、增强的 DMA 子系统及动态功耗管理功能，能够支持复杂的控制和信号处理任务，同时保持了极高的数据吞吐量。ADSP-BF561 功能模块框图如图 3.6 所示。

ADSP-BF561 是对流行的 Blackfin 处理器家族的功能扩展，特别适合于很多不同的工业、仪器仪表、医学和消费类应用，如 VoIP、多媒体录放设备、多媒体附件、网络音频、工业控制、成像等。它能够根据需要的数据带宽进行伸缩，并且允许混合控制和信号处理功能。它的处理器内核最高频达 600MHz（1200MMACS），提供了 328KB 的片上内存，其中每个内核有 32KB 的 L1 指令内存、64KB 的 L1 数据内存、4KB 的临时内存，另外两个内核共享 128KB 的低延时 L2 内存。它还有两个支持 ITU-R BT.656 视频数据格式的 PPI 接口，能支持视频数据的并行输入和输出。此外，它支持基于硬件的知识产权保护、12 个双通道全双工同步串口（支持 8 个立体声通道）、12 个支持一维和二维数据传输的外设 DMA 通道和 10/100MII 以太网接口，内存控制器可无缝连接多个外部 SDRAM、SRAM、Flash 或 ROM 内存组。

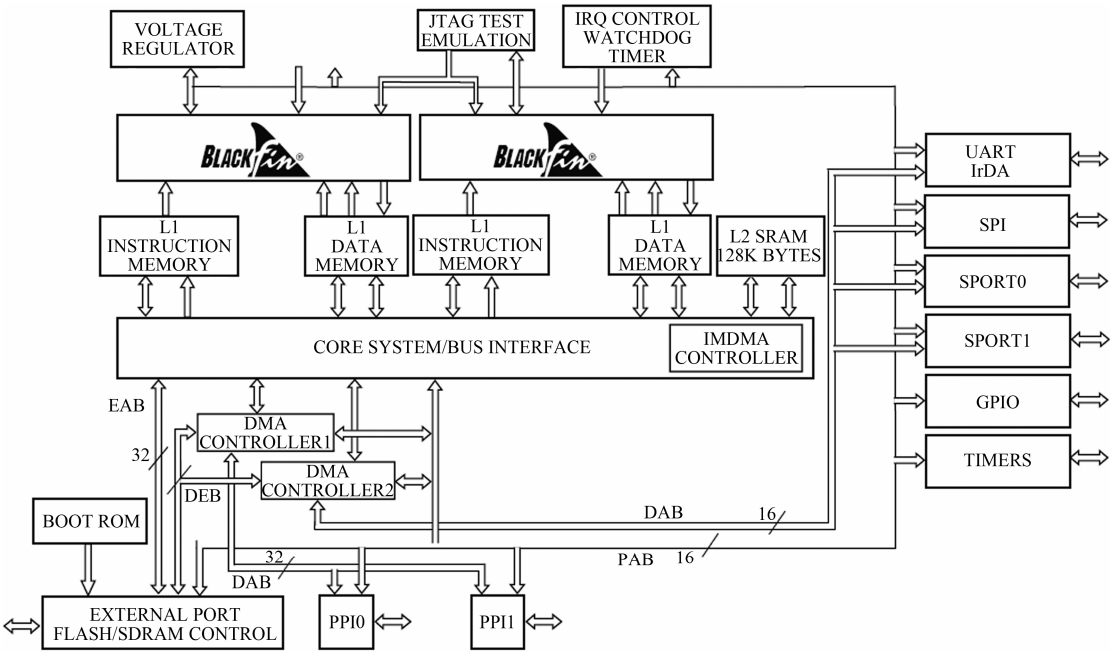


图 3.6　ADSP-BF561 功能模块框图

3.3　Blackfin 处理器架构

3.3.1　Blackfin 处理器架构概述

Blackfin 处理器基于由 ADI 和 Intel 公司联合开发的微信号架构（MSA），将 32 位 RISC 型指令集和双 16 位乘法累加（MAC）信号处理功能与通用型微控制器所具有的易用性组合

在一起。该组合使 Blackfin 处理器能够在信号处理和控制处理应用中均发挥上佳的作用——在许多场合中免除了增设单独异类处理器的需要，极大简化了硬件和软件设计任务。

目前 Blackfin 处理器在单内核产品中可提供高达 756MHz 的性能。Blackfin 处理器系列中的新型对称多处理器成员（ADSP-BF561）在相同的频率条件下实现了性能的翻番。Blackfin 处理器系列还提供了低至 0.8V 的业界领先功耗性能。为满足当今及未来的信号处理应用（包括宽带无线电、具有音频/视频功能的因特网工具和移动通信），这种高性能与低功耗的组合是必不可少的。

Blackfin 处理器为系统设计师提供了重要的优势，包括可实现各种新型市场和应用的高性能信号处理和高效控制处理能力，可令系统设计师使器件功耗模式与终端系统要求相适应的动态电源管理（DPM）能力，以及可确保产品开发时间最小化的易用型混合 16/32 位指令集架构和开发工具套件。

其他主要优点包括如下。

1．高性能处理器内核

Blackfin 处理器架构基于一个 10 级 RISC MCU/DSP 流水线和一个专为实现最佳代码密度而设计的混合 16/32 位指令集架构。Blackfin 处理器架构完全符合 SIMD（单指令多数据）标准，包含专门用于加速视频和图像处理的指令。该架构很适合于全信号处理和分析，还可在单内核或双内核器件上提供高效 RISC MCU 控制任务执行能力。由于具有最佳代码密度且只需进行极少（或者完全不需要进行）代码优化处理，可缩短产品的面市时间，而不会遇到其它传统处理器所常见的性能空间障碍。

2．高带宽 DMA 能力

所有 Blackfin 处理器均具有多个独立的 DMA 控制器，这些控制器支持自动数据传输，所需处理器内核开销极少。DMA 传输可出现在内部存储器和诸多具有 DMA 功能的外设之间。传输也有可能出现在外设和与外部存储器接口相连的外部器件（包括 SDRAM 控制器和异步存储器控制器）之间。

3．视频指令

除了具有对 8 位数据及许多像素处理算法所常用字长的固有支持外，Blackfin 处理器架构还包含了专为增强视频处理性能而定义的指令。例如，离散余弦变换（DCT）通过 IEEE 1180 舍入操作得到支持，而“SUM ABSOLUTE DIFFERENCE”（绝对差之和）指令则支持在诸如 MPEG2、MPEG4 和 JPEG 等视频压缩算法中所使用的运动估计算法。

利用软件实现视频压缩算法使 OEM 制造商能够在不变更硬件的情况下适应不断发展的标准和新功能。增强型指令可使 Blackfin 处理器在那些先前主要是由 ASIC 媒体处理器或硬连线芯片组来满足的应用中一试身手。归根结底，Blackfin 处理器将在帮助降低系统总成本的同时使终端应用产品上市时间得以缩短。

4．高效控制处理

Blackfin 处理器架构还具有各种在 RISC 控制处理器中最为常见的优点。这些优点包括功能强大且灵活的分层存储器架构、出众的代码密度及各种各样的微控制器型外设，包括 10/100 以太网 MAC、UART、SPI、CAN 控制器、支持 PWM 的定时器、看门狗定时器、实时时钟及一个无缝的同步和异步存储器控制器等。所有这些优点为设计师提供了巨大的设计灵活性，并最大限度地降低了终端系统成本。

5．分层存储器

Blackfin 处理器存储器架构在器件实现中提供了 L1 和 L2 存储模块，如图 3.7 所示。L1 存储器直接与处理器内核相连，以全系统时钟频率运行并为实时算法程序段提供了最大的系统性能。L2 存储器是一种较大的大容量存储模块，其性能虽略有下降，但运行速度仍然高于片外存储器。L1 存储器架构的实现旨在提供信号处理所需的性能及通用型微控制器所拥有的编程简易性。这是通过允许将 L1 存储器配置为 SRAM、Cache 或两者的组合实现的。通过支持 SRAM 和 Cache 编程模型，系统设计师能够把要求高带宽和低延迟的关键实时信号处理数据组分配至 SRAM 中，而将更多的“软”实时控制/OS 任务存储于高速缓冲存储器上。

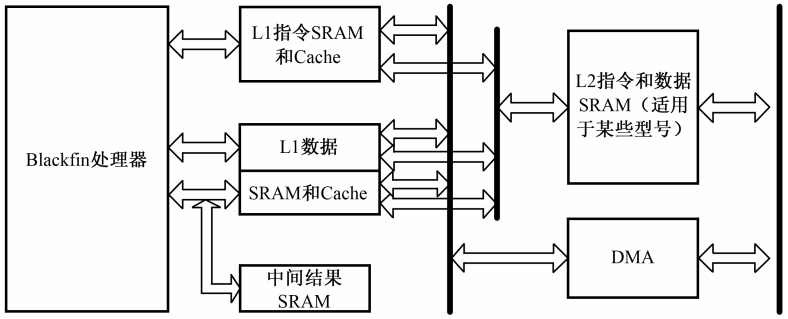


图 3.7　Blackfin 处理器存储模块设计

存储器管理单元（MMU）规定了存储器保护格式，当其与内核用户及监控模式相组合时，就能够支持一个全实时操作系统（RTOS）。该 RTOS 运行于监控模式，并对存储模块及其他系统资源进行分割，以便实际应用程序运行于用户模式之中。这样 MMU 就提供了一种用于实现完善系统和应用的隔离而安全的环境。

6．出众的代码密度

Blackfin 处理器架构支持多长度指令编码。使用频率非常高的控制型指令被编码为紧致 16 位字，而更多的算术密集型信号处理指令则被编码为 32 位。处理器将把 16 位控制指令与 32 位信号处理指令加以混合和链接，以形成 64 位组，从而实现存储器存储密度的最大化。当进行指令高速缓存和取指令操作时，内核将自动对总线的长度进行充分的压缩，因为它没有对准方面的限制。当组合起来使用时，这两种功能将使 Blackfin 处理器提供出堪与

业界领先的 RISC 处理器相媲美的性能。

7．动态电源管理

所有 Blackfin 处理器均采用多种节能技术。它基于一种选通时钟内核设计，可按照逐条指令来选择性地切断功能单元的电源。Blackfin 处理器还支持多种针对所需 CPU 动作极少（或根本不需要 CPU 动作）的断电模式。最重要的一点是，Blackfin 处理器支持一种自含动态电源管理电路，借助该电路即可对工作频率和电压进行独立控制，以满足正在执行的算法的性能要求。这些转换可以在一个 RTOS 或用户固件的控制之下连续出现。大多数 Blackfin 处理器都提供了片上内核稳压电路，可在低至 0.8V 的电压条件下工作，因而特别适合于需要延长电池使用寿命的便携式应用。

8．系统集成度高

系统具有丰富的外设集合，通过几个高带宽总线连接到处理器核上，提供系统配置的灵活性及优秀的整体系统性能。通用外设包括 UART、定时器（具有 PWM－脉冲宽度调制和脉冲测量能力）、通用 I/O 引脚、实时时钟、看家狗定时器。这组功能满足很大范围的一些典型系统支持，并被该部件的扩充能力所放大。除了这些通用外设，处理器还包含高速串口和并口，能够与各种音频、视频和 MODEM 编解码器功能提供接口，一个终端控制器可以灵活地管理片上或外部的各种中断，功率管理控制功能可以针对不同应用场景提供合适的处理器功率特性。这样用户就可以结合工业标准接口和高性能信号处理核快速开发有成本效益的方案，而无需昂贵的外部组建。

9．易用性

许多过去需要同时采用一个高性能信号处理器和一个单独的高效控制处理器的应用，现在只需采用一个 Blackfin 处理器就足够了。这极大地缩减了开发时间和成本，并最终加快了终端产品的面市。此外，由于只需采用一组开发工具，因而减少了系统设计师的学习时间。

3.3.2　Blackfin 处理器内核基础知识

Blackfin 处理器包括一个具有 10 级 RISC MCU/DSP 流水线的高性能 16/32 位嵌入式处理器内核、用于实现最佳代码密度的可变长度指令集构架以及具有面向加速视频和多媒体处理的指令的全 SIMD 支持。结合图 3.8 中 ADSP-BF533 的内核结构，现对 Blackfin 处理器内核说明如下。

1．通用寄存器文件

Blackfin 处理器内核包括一个用于通用运算单元的 8 路×32 位数据寄存器文件。支持 8/16/32 位有符号/无符号整数，以及 16/32 位带符号分数。在每个时钟周期支持两个 32 位读

出和两个 32 位写入操作。还可将其作为一个 16 路×16 位数据寄存器文件来进行存取。

除了支持循环缓冲以外，地址寄存器文件还提供了通用型寻址机制。该寄存器文件由 6 个表目组成，另外还包括一个帧指针和一个栈指针。帧指针可用于子例程参数传输，而栈指针则可用于存储子程序调用的返回地址。

2．数据运算器

数据运算器包括 2 个 16 位 MAC，2 个 40 位 ALU，3 个 8 位视频 ALU，1 个桶形移位器。所有这些都能够处理来自数据寄存器文件（R0～R7）的 8/16/32 操作数。可将每个寄存器作为一个 32 位寄存器或一个 16 位寄存器的高位部分或低位部分来进行存取。MAC 可在每个周期执行一个 16 位乘法，并将结果累加到 40 位累加器。同时支持有符号和无符号格式、舍入操作及饱和操作等。

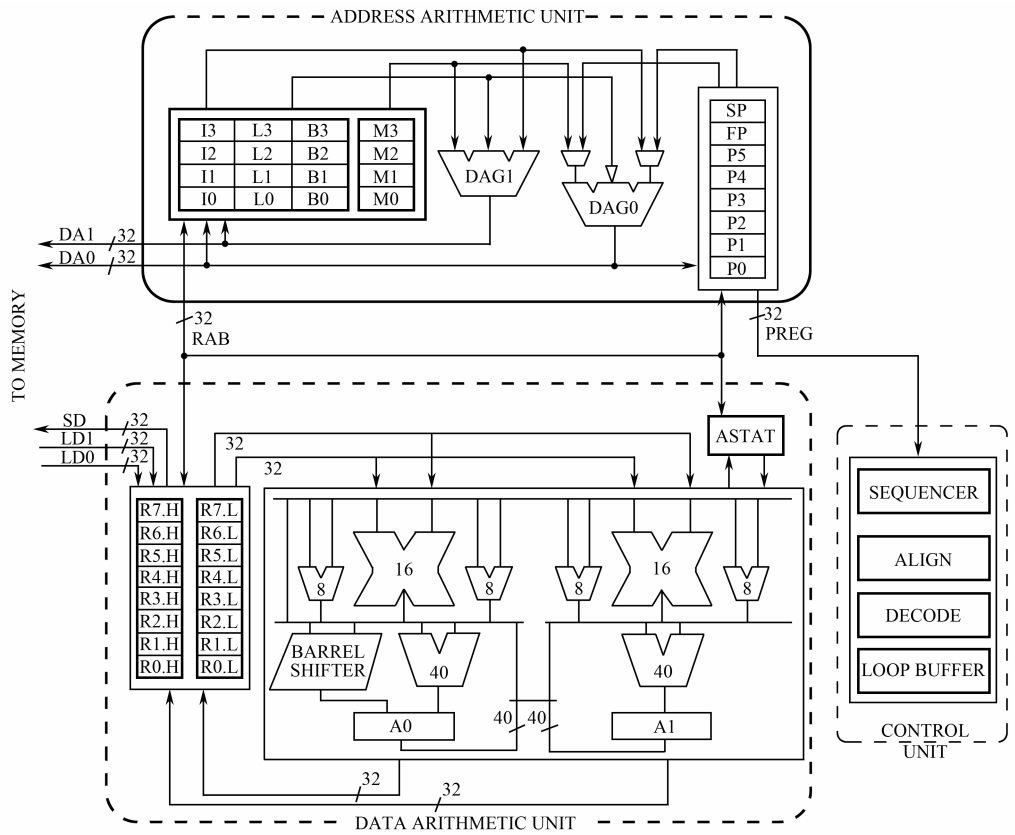


图 3.8　ADSP-BF533 处理器内核

在单个时钟周期，SIMD 架构能够对多达两个 32 位值进行读出和写入操作。然而，由于能够对 R0～R7 寄存器的高位和低位部分进行独立寻址（Rx、Rx.H 或 Rx.L），因此每个计算模块都能从两个 32 位输入值和 4 个 16 位输入值中进行选择，且并未对输入数据加以限制。计算结果可作为 32 位实体或寄存器的高或低 16 位部分重新写入寄存器文件。此外，

累加的方法有可能因为数据通路的不同而存在差异。例如，A0 可能是一个恒定加法，而 A1 则可能是一个恒定减法。这种能力被称为“灵活的 SIMD”。

两个累加器的长度均为 40 位，从而提供了 8 位扩展精度。与通用型寄存器相似，两个累加器均能够以 16/32/40 位的增量进行存取。Blackfin 架构还支持可生成两个 16/32/40 位结果或 4 个 16 位结果的组合型加法/减法指令。在希望获得 4 个 16 位结果的场合，高低位部分结果可互换。这是项非常强大的功能，比如它能够显著改善 FFT 基准程序的结果。

3．地址运算器

两个数据地址发生器（DAG0 和 DAG1）提供了用于实现存储器同时双重操作数读取的地址。两个 DAG 共用一个包含了 4 组 32 位索引（I）、长度（L）、基数（B）和修改（M）寄存器的寄存器文件。另外还有 8 个附加的 32 位地址寄存器（即 P0～P5、帧指针和栈指针），它们可被用做针对变量和栈位置的通用标引的指针。

4 组 I、L、B 和 M 寄存器可用于实现循环缓冲。当一起使用时，每组索引、长度和基数寄存器（如 I0、L0、B0、M0）都能够在内部或外部存储器中实现一种独特的循环缓冲器。Blackfin 架构还支持各种寻址模式，包括间接型、自动增量和减量型、索引型及位反转型。最后，所有地址寄存器的长度均为 32 位，从而可支持 Blackfin 处理器架构全部 4G 字节的地址范围。

4．程序定序器单元

程序定序器负责控制指令执行的流程，并支持条件转移和子例程调用及嵌套式零开销循环。一个多级全互锁型流水线可确保代码按照预期的方式来执行，并将所有的数据故障与编程装置隔离开来。此类流水线通过在必要时停转的方法确保了结果的准确度，以获得正确结果。这极大地简化了编程任务，因为软件工程师无需彻底了解流水线延迟问题。片上互锁硬件可确保操作数据在一个特殊指令的执行过程中处于有效状态。

除了有限的多种 64 位指令程序包之外，Blackfin 架构还支持 16 位和 32 位指令长度。这通过把最常用的控制指令编码为紧致 16 位字，并将更加棘手的数学运算编码为 32 位双字的方法确保了最大的代码密度。

5．操作模式

Blackfin 内核结构提供 3 种操作模式：用户模式、超级用户模式及仿真模式。用户模式对一些系统资源的访问是受限的，提供了受保护的软件环境。超级用户模式对系统和内核资源的访问是不受限的。

3.3.3　数据运算指令简介

下面利用 Blackfin 汇编指令描述算术逻辑运算，它使用代数化语法，简化了汇编代码的开发。

1．单个 16 位加减运算

任意两个 16 位寄存器中的数据可以相加、减而产生一个 16 位的运算结果，并将结果存储到另外一个 16 位寄存器中。例如，R3.H = R1.L + R2.H(ns)就是将 R1.L 和 R2.H 中的值相加，结果存入 R3.H 中，其中（ns）表示结果不进行饱和操作。

2．双 16 位加减运算

任意两个 32 位寄存器可用来存储双 16 位加减运算的 4 个输入数据，并且两个 16 位的运算结果可以存放在一个 32 位寄存器中。例如，R3 = R1 +|–R2(s)，就是将 R1 和 R2 的高 16 位相加，同时将低 16 位相减，运算结果进行饱和运算（s 代表进行饱和运算），将最终结果存入 R3 寄存器的高 16 位和低 16 位中。

3．4 个 16 位加减运算

在 4 个 16 位加减运算中，只能用两个相同的 32 位寄存器存储 4 个 16 位的输入数据来完成 4 元加减法。也就是说，在两对相同的 16 位寄存器中可以进行两个运算。例如，R3 = R1 +|– R2, R4 = R1 –|+ R2，表示 R1 和 R2 的高低 16 位分别相加和相减，注意，用符号“,”可以将同一个指令周期中的两个指令分开。

4．单个 32 位运算

单个 32 位加减运算产生 32 位结果，存储到其他的 32 位寄存器中。例如，R3=R1+R2，表示 R1 与 R2 相加，结果存储到 R3 中。

5．双 32 位运算

可以扩展到双 32 位加减运算，类似于 16 位运算，只是输入操作数据和输出结果都是 32 位。例如，R3 = R1 + R2, R4 = R1−R2，表示 R1 和 R2 同时相加和相减，相加结果存储到 R3 中，相减结果存储到 R4 中。

在以上 ALU 运算中，加减法数据尺度都是 16 位或 32 位。由于用来存储运算结果的字长有限，因此可能造成溢出。解决办法是在运算指令中根据操作数长度，将运算结果饱和处理到 16 位或 32 位。例如，16 位字运算经过饱和运算都不会超过 16 位有符号数所能表达的最大正数和最小负数。表 3.1 列出了可供 ALU 操作使用的模式和选项。

6．32 位 ALU 逻辑运算

例如，R3 = R1 & R2 表示 R1 和 R2 寄存器按位进行逻辑与操作，将结果存储到 R3 中。此外还有或运算 R3=R1|R2，非运算 R2=～R1，异或运算 R3=R1^R2 等。注意，在 Blackfin 中没有 16 位逻辑运算。

表 3.1　ALU 操作的运算模式和选项

模　　式	选　　项	例子与解释
双 16 位和 4 个 16 位运算（opt_mode_0）	S	结果饱和处理为 16 位 R3=R1+|–R2(s)
	CO	在复数运算中，交换目的寄存器中的运算结果的次序：R3=R1+|–R2(co)
	SCO	S 与 CO 的结合
双 32 位和 40 位运算（opt_mod_1）	S	结果饱和处理为 32 位 R3=R1+R2, R2=R1–R2(s)
4 个 16 位运算（opt_mode_2）	ASR	算术右移，在结果存入目的寄存器前将其减半 R3=R1+|–R2, R4=R1–|+R2(s,asr) 结果在饱和处理前进行缩放
	ASL	算术左移，在结果存入目的寄存器前将其翻倍

7．单个 16 位乘法运算

任意两个 16 位寄存器相乘得到一个 32 位的结果，可以存储到累加器或数据寄存器中，如 R3=R1.L*R2.H。16 位的结果也可以存储到一个半字寄存器中，如 R3.H=R1.L * R2.H。

8．单个 16 位乘/累加运算

例如，指令 A0+=R1.L*R2.L 表示 R1.L 和 R2.L 中的数据相乘，结果与累加器 A0 中的数据相加，最终结果存储到 A0 中。此外，也可以将最终结果存储到一个半字寄存器中。如 R5.L=(A0+=R1.L*R2.L)，该指令表示将累加器中的结果截取成 16 位（如忽略 A0 的低 16 位），并将 A0 的高 16 位存储到 R5 寄存器的低 16 位中。

9．使用两个累加器实现双 16 位乘法运算

例如，A1=R1.H*R2.H, A0=R1.L*R2.L，两对输入数据存储在两个寄存器 R1 和 R2 中，两个 32 位结果分别存储到累加器 A0 和 A1 中。

10．使用两个 32 位寄存器实现双 16 位乘法运算

当用两个 32 位寄存器来完成双 16 位乘法运算时，32 位目的寄存器必须成对使用，如 R0:R1、R2:R3、R4:R5 或 R6:R7。例如，指令 R0=R2.H*R3.H, R1=R2.L*R3.L，表示双 32 位结果分别存储到 R0 和 R1 中。因此，R0 和 R1 必须成对地用于存储乘积的高字和低字。其他寄存器对，如 R4:R5 也可用来代替上例中的 R0:R1。

11．使用两个 16 位目的寄存器实现双 16 位乘/累加运算

可以通过扩展实现双 MAC 运算使运算能力加倍，如 A1–=R1.H*R2.H, A0+=R1.L*R2.L。此外，双 MAC 运算的结果可以存储到两个 16 位寄存器中，如 R3.H=(A1–=R1.H*R2.H), R3.L = (A0+=R1.L*R2.L)。注意，A1 中的结果必须存储到 R3 的高半字中，A0 中的结果必须存储到 R3 的低半字中。

12．使用两个 32 位目的寄存器实现双 16 位乘/累加运算

这种情况下，双 16 位乘/累加运算中的 32 位目的寄存器必须以成对形式保存数据，如 R0:R1。如 R0=(A0+=R2.H*R3.H), R1=(A1+=R2.L*R3.L)。注意，A1 与高字寄存器有关（本例中的 R1），A0 与低字寄存器有关（R0）。以上的乘法和乘/累加运算在默认条件下操作是无选项的。默认选项意味着输入数据是有符号小数。Blackfin 处理器可以处理不同格式的数据。所有可能的选项及其说明如表 3.2 所示。

表 3.2　16 位乘法选项

选　　项	说　　明
默认值（无选项）	输入数据操作数是有符号小数
（FU）	输入数据操作数是无符号小数，无移位校正
（IS）	输入数据操作数是有符号整数，无移位校正
（IU）	输入数据操作数是无符号整数，无移位校正
（T）	输入数据操作数是有符号小数，当结果复制到目的半字寄存器时，截去累加器值的低 16 位
（TFU）	输入数据操作数是无符号小数，当结果复制到目的半字寄存器时，截去累加器值的低 16 位

Blackfin 处理器的另一个特点是，它包含 4 个附加的 8 位视频 ALU。在图像和视频处理应用中，提供了专用的视频指令集合用于视频 ALU。由于数据寄存器是 32 位的，4 个 8 位运算（如加、减、平均、绝对值等）可用单个指令执行。

3.3.4　地址运算指令简介

地址运算单元的主要功能是为访问 Blackfin 处理器的存储器产生地址。Blackfin 处理器按字节寻址。通过 P0～P5 可按 8、16 或 32 位寻址；16 位和 32 位数据存取可以通过 I0～I3；32 位则可以通过堆栈指针寄存器和帧指针寄存器。Blackfin 寻址方式主要有以下几种。

1．间接寻址

指令中的中括号表示在数据加载/存储操作中使用的地址指针，即变址指针和栈指针/帧指针寄存器。例如，指令 R0=[P0]表示指针寄存器 P0 中的值为一个地址，本指令的作用是将该地址对应的内存中的值（32 位）加载到数据寄存器 R0 中。

而[P1]=R0 表示存储操作，将数据寄存器 R0 中的值（32 位）存储到内存中以 P1 中的值指向的存储器位置上。

2．支持 16 位和 8 位数据存取的间接寻址

上面的例子是针对 32 位数据操作的。地址运算还支持 16 位和 8 位加载/存储操作。16 位操作指令为 R0=W[P0](z)，其中前缀 W 表示字存取（16 位），后缀(z)表示高位必须以零填充。8 位加载操作指令为 R0=B[P0](x)，前缀 B 表示字节存取，后缀(x)表示高位要以符号位扩展。

3．后修改地址的寻址方式

地址指针寄存器 P0～P5 和变址寄存器 I0～I3 可以在加载/存储操作完成后才修改和更新其内容。该方式在加载或存储下一个数据时，可以自动更新地址指针，使它指向下一个存储单元，非常有用。后修改地址可以用++或--实现，分别表示加和减运算。例如，R0=[P0++]表示完成加载操作后，将 P0 值加 4。而对于 R0=W[P0++](z)，由于加载的是一个字，因而需要加 2 字节。同样的，如果 R0=B[P0++](x)，P0 则需要增加 1 字节。

同样，用后修改地址方式进行 32 位存储操作，可描述为[P1--]=R0。这种情况下，P1 在存储操作后将被减去 4。

4．预修改地址寻址方式

该方式仅在指向堆栈指针时适用。例如，[--SP]=R0 中，表示在把 R0 存储到堆栈寄存器操作前，先将堆栈指针 SP 减 4。与之相反，出栈指令将堆栈中的内容加载到特定寄存器，并对堆栈指针进行后修改地址操作，如 R0=[SP++]。

5．修改指针寄存器寻址方式

有些情况下，需要通过增加或减少一个字长来修改指针，可以通过简单的立即数来修改指针。例如，指令 R1=[P0+=0x08]，在地址 P0+0x08 处加载数据到 R1 寄存器。注意，这个操作中不改变指针寄存器的预修改地址寻址，操作后 P0 的值保持不变。

如果操作后需要修改指针寄存器，指针寄存器可被另一个指针寄存器修改（如 R0=[P0++P1]，P1 为偏移值）。同样的，变址寄存器也可以通过 M 寄存器来修改。这种寻址方式支持增量修改和减量修改两种方式。例如，指令 R1=[I0++M0]，操作后 I0 寄存器的值会增加（增加量为 M0 中的值）。

除了以上线性寻址方式外，Blackfin 处理器还提供循环寻址方式。循环缓冲区中的地址产生器连续递进，当指针到达循环缓冲区末端时，地址产生器会自动回绕到起始地址。循环缓冲区由长度寄存器（L）、基址寄存器（B）、修改寄存器（M）和变址寄存器（I）控制。

3.3.5　Blackfin 内存结构

以 ADSP-BF533 处理器为例，系统将内存看做一个统一的 4G 字节的地址空间，使用 32 位地址。所有资源，包括内部存储、外部存储和 I/O 控制寄存器等，都在该空间上占用各自的片段。存储器部分采用分级内存结构，以提供良好的成本/性能平衡：既有快速、低响应延迟的片上内存（如 Cache 或 SRAM），又有大容量、低成本、低性能的片外存储器系统。

L1 内存系统是 Blackfin 处理器主要的最高性能的内存。片外存储系统通过外部总线接口单元（EBIU）访问，提供对 SDRAM、闪存和 SRAM 的扩充，可任意访问高达 132M 物理字节。内存 DMA 控制器提供高带宽的数据移动能力，它可以在内、外部存储器空间之间执行大块数据或代码的传输。

1．内部（片上）存储器

处理器有 3 块片上内存提供到内核的高带宽存取。第一块是 L1 指令内存，包含最高 80K 字节的 SRAM，其中 16K 字节可以设置为 4 路集合相关联的 Cache。第二块是 L1 数据内存，包含一排或两排最高 32K 字节的内存块。内存组是可配置的，可提供 Cache 或 SRAM 功能。这两个内存块以处理器全速度访问。第三块是 4K 字节的中间结果 SRAM，与 L1 内存运行速度相同，但只能作为数据 SRAM 访问，不能配置为 Cache。

2．外部（片外）存储器

外部存储器通过外部总线接口单元（EBIU）访问。该 16 位接口能够连接一组 SDRAM，以及最多 4 组异步存储器设备——包括 Flash、EPROM、ROM、SRAM 和内存映射 I/O 设备。兼容 PC133 的 SDRAM 控制器可以与最高 128M 字节的 SDRAM 连接。SDRAM 控制器允许同时打开每个内部 SDRAM 排（最多 4 个组）中的一个，能够提高整体系统性能。异步内存控制器可以最多控制 4 组设备，定时参数灵活，设备种类多样。每组占用 1M 字节的片段，不管所使用的设备的尺寸，所以这些组只有在每个设备都用完 1M 内存时才会变成是紧邻的。

3．I/O 内存空间

Blackfin 处理不定义单独的 I/O 空间。所有资源都映射到扁平的 32 位地址空间。片上 I/O 设备将其控制寄存器映射到内存映射寄存器（MMR），在靠近 4G 字节地址空间的顶部。它们分为两个更小的块，一个包含对所有内核功能的控制 MMR，另一个包含用于设置和控制内核外片上外设的寄存器。MMR 只在超级用户模式下访问，对片上外设而言是预留的。

ADSP-BF533 内部/外部内存映射如图 3.9 所示。

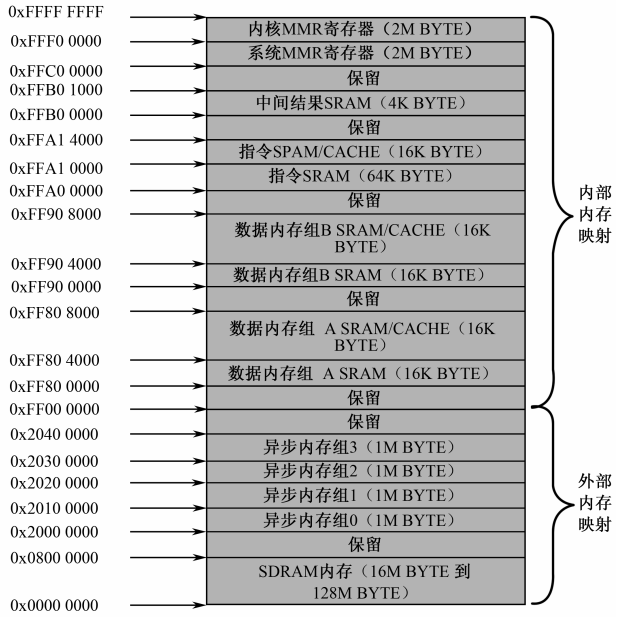


图 3.9　ADSP-BF533 内部/外部内存映射

3.3.6　事件处理

内核中事件控制器处理所有发到处理器的异步和同步事件。ADSP-BF533 处理器提供的事件处理支持嵌套式处理和优先级。嵌套允许多个事件服务函数可以同时是活动的。优先级确保高优先级的事件比低优先级的事件先得到处理。控制器支持 5 种类型的事件：

（1）仿真：该事件使处理器进入仿真模式，允许通过 JTAG 接口下达对处理器的命令和控制。

（2）复位：该事件使处理器复位。

（3）不可屏蔽中断（NMI）：可由软件看家狗计数器或 NMI 输入信号产生。NMI 事件经常用于下电指示，以启动有序的系统下电操作。

（4）异常：与程序流同步发生（即异常发生在指令被允许结束前）。发生异常的情况包括数据对齐违规、未定义指令等。

（5）中断：其发生与程序流是异步的，可以由输入端脚、定时器和其他外设产生，也可以显式地由软件指令产生。

每种事件类型都有一个相关的寄存器来保存返回地址和相关的“从事件返回”指令。当事件被触发后，处理器状态被保存在管理者（Supervisor）堆栈。ADSP-BF533 处理器的事件控制器包含两个级别：核心事件控制器（CEC）和系统中断控制器（SIC）。CEC 与 SIC 联合对所有系统事件确定优先级并进行控制：来自外设的中断先进入 SIC，然后直接被发送到 CEC 的通用中断。

1．内核事件控制器（CEC)

除专用中断和异常事件外，CEC 还支持 9 个通用中断（IVG15-7），其中两个最低优先级的（IVG15-14）建议留作软件中断处理程序，另外 7 个用来支持外设的中断。表 3.3 描述了到 CEC 的各种输入，在事件向量表（EVT）中确定了它们的名称，并列出其属性。

表 3.3　核心事件控制器（CEC）

优先级（0 最高）	事 件 类 别	EVT 项（入口）
0	仿真/测试控制	EMU
1	重启	RST
2	非屏蔽中断	NMI
3	异常	EVX
4	保留	
5	硬件错误	IVHW
6	内核定时器	IVTMR
7	通用中断 7	IVG7
8	通用中断 8	IVG8
9	通用中断 9	IVG9

续表

优先级（0 最高）	事 件 类 别	EVT 项（入口）
10	通用中断 10	IVG10
11	通用中断 11	IVG11
12	通用中断 12	IVG12
13	通用中断 13	IVG13
14	通用中断 14	IVG14
15	通用中断 15	IVG15

2．系统中断控制器（SIC）

SIC 将来自外设中断源的事件映射为区分优先级的通用中断并发送到 CEC。尽管处理器提供了默认映射，但用户可以通过在中断分配寄存器（SIC_IARx）中写入恰当的值改变中断事件的映射和优先级。表 3.4 描述了 SIC 的输入及其到 CEC 的默认映射。

表 3.4　系统中断控制器（SIC）

外设中断事件	默 认 映 射
PLL 唤醒	IVG7
DMA 错误	IVG7
PPI 错误	IVG7
SPORT 0 错误	IVG7
SPORT 1 错误	IVG7
SPI 错误	IVG7
UART 错误	IVG7
实时时钟	IVG8
DMA 通道 0（PPI）	IVG8
DMA 通道 1（SPORT 0 接收）	IVG9
DMA 通道 2（SPORT 0 发送）	IVG9
DMA 通道 3（SPORT 1 接收）	IVG9
DMA 通道 4（SPORT 1 发送）	IVG9
DMA 通道 5 （SPI）	IVG10
DMA 通道 6（UART 接收）	IVG10
DMA 通道 7（UART 发送）	IVG10
定时器 0	IVG11
定时器 1	IVG11
定时器 2	IVG11
Port F GPIO 中断 A	IVG12
Port F GPIO 中断 B	IVG12
内存 DMA 流 0	IVG13
内存 DMA 流 1	IVG13
软件看门狗定时器	IVG13

3．事件控制

处理器提供了非常灵活的机制来控制事件的处理。其中 CEC 用 3 个寄存器来协调和控制事件，每个寄存器为 32 位。

（1）CEC 中断锁存寄存器（ILAT）。ILAT 寄存器指示事件何时被锁存。当处理器锁存该事件后，对应位被置位，而当事件被系统接收后该位被清除。该寄存器由控制器自动更新，但也可通过写操作来取消锁存事件。该寄存器在超级用户模式下是可读的，且只有在超级用户模式下并且对应的 IMASK 位被清除时才能写入。

（2）CEC 中断屏蔽寄存器（IMASK）。IMASK 寄存器控制单个事件的屏蔽和非屏蔽。当 IMASK 寄存器中的某位被置位，该事件是非屏蔽的，当激活时会被 CEC 处理。IMASK 寄存器中被清除的位表示对应事件被屏蔽了，处理器不会对该事件服务，即使该事件在 ILAT 寄存器中被锁存。该寄存器在超级用户模式下可读写。通用中断可使用 STI 和 CLI 指令进行全局使能和禁止。

（3）CEC 中断挂起寄存器（IPEND）。IPEND 寄存器跟踪所有嵌套事件。IPEND 寄存器的置位操作表示对应事件当前是活动的或处于某级嵌套。该寄存器被控制器自动更新，在超级用户模式下是可读的。

SIC 通过 3 个 32 位中断控制和状态寄存器提供更多事件控制能力。每个寄存器对表 3.4 中每个外设中断事件都有对应的位。

（1）SIC 中断屏蔽寄存器（SIC_IMASK）。该寄存器控制每个外设中断事件的屏蔽状态。寄存器中某位置位时，对应外设事件是非屏蔽的，当激活时可被系统处理。该位清除则将对应外设事件屏蔽，阻止处理器为该事件提供服务。

（2）SIC 中断状态寄存器（SIC_ISR）。多个外设可以映射为一个事件，该寄存器允许软件确定哪个外设事件触发了该中断。某位置位表示对应外设激活了该中断，清除则表示对应外设没有激活该中断。

（3）SIC 中断唤醒使能寄存器（SIC_IWR）。通过使能该寄存器中的对应位，外设可以配置为可以唤醒处理器——如果事件产生时内核是空闲的。

因为多个中断源可以映射到一个通用中断，多个脉冲激活可能同时发生，此时处理器可能正在（或还没有）处理该中断输入上已经监测到的中断事件。IPEND 寄存器的内容由 SIC 作为中断确认来监控。当检测到中断上升沿时，适当的 ILAT 寄存器位被置位。当各自的 IPEND 寄存器位被置位后，该位被清除。IPEND 置位表示对应事件已经进入处理器流水线。此时 CEC 在对应的事件输入上识别下一个上升沿事件并排队。

3.3.7　DMA 控制器

ADSP-BF533 处理器有多个相互独立的 DMA 通道，可以用最小的处理器内核开销支持自动数据传输。DMA 传输可以发生在处理器内部存储器和任何支持 DMA 的外设之间。另外，DMA 传输还可以在任何支持 DMA 的外设和连接到外部存储器接口的外部设备之间实

现，包括 SDRAM 控制器和异步存储器控制器。支持 DMA 的外设包括 SPORT、SPI 端口、UART 和 PPI。每个支持 DMA 的外设具有至少一个指定的 DMA 通道。

DMA 控制器支持一维和二维 DMA 传输。DMA 传输初始化可以用寄存器实现，也可以用被称做描述符块的几组参数实现。二维 DMA 能够支持任意行和列尺寸（最大 64KB×64KB 个元素），以及任意的行和列步长（最大±32KB 个元素）。而且，列的步长可以小于行的步长，允许隔行数据流的实现。该特征对视频应用特别有用，因为数据可以及时完成隔行操作。

控制器支持的 DMA 类型的例子包括单个线性缓冲区，完成时即结束；循环自动更新缓冲区，每次填满或部分填满时产生中断；一维或二维 DMA，使用链接的描述符列表；使用描述符数组的二维 DMA，在一个通用页上只指定 DMA 基地址。

基于描述符的 DMA 传输在发起 DMA 传输序列时，需要一组存储在存储器中的参数。这类传送允许将多个 DMA 序列连接在一起。在基于描述符的 DMA 传输中，一个 DMA 通道可以被编程建立，并且在当前序列完成之后自动启动另外一个 DMA 传输。基于自动缓存的 DMA 传输允许处理器直接编程 DMA 控制寄存器，以发起一个 DMA 传输。传输完成时，控制寄存器就被其原始设定值自动更新。

除了指定的外设 DMA 通道，有两对内存 DMA 通道，用于在处理器系统这两个不同的内存之间传输数据。这使得数据块可以在任何内存间传输——包括外部 SDRAM、ROM、SRAM 和闪存，该传输过程只需要最小的处理器介入量。内存 DMA 传输可以用非常灵活的基于描述符的方法，或利用标准的基于寄存器的自动缓冲区机制来控制。

1．基于描述符的 DMA 传输

基于描述符的 DMA 传输是 Blackfin 控制 DMA 传输的最普通方法。使用该方法时，DMA 通道需要一组称为 DMA 描述符的参数，这些参数存储在存储器中。每个描述符包含一个特定 DMA 传输序列所需的所有信息，其组成结构如下：

（1）要传输数据块的 32 位起始地址。

（2）要传输的数据量。

（3）其他的各种控制信息，如该 DMA 通道做什么，传输何时完成等配置信息。

（4）指向下一个描述符的指针

DMA 描述符中各元素的定义如表 3.5 所示，其中 BASE 是描述符的起始地址。

表 3.5　DMA 描述符参数

地　　址	地址内容名称	描　　述
BASE+0	DMA 配置字	描述符所有者、DMA 配置、完成状态
BASE+2	DMA 传输计数	需要传送的数据元（8/16/32 字）数目
BASE+4	DMA 起始地址	传输起始地址的低 16 位
BASE+6	DMA 起始地址	传输起始地址的高 16 位
BASE+8	下一个描述符指针	下一个描述符块的头地址的低 16 位

处理器可以将标识多个传输的多组描述符放入存储器中，并构成一个链表。当一个链表被生成时，DMA 通道就有了完成多个传输序列所需的所有信息，这些传输序列不需要处理器干预。

到一个传输序列结束时，下一个描述符指针必须指向存有一个 16 位数据的存储单元，该 16 位数据的第 15 位为 0。如果该数据的第 0 位为 1，则 DMA 通道仍能被使用，但被延迟，并且剩余的 FIFO 值不被丢弃。

表 3.6 描述了 DMA 配置字中的各个位，该寄存器位于描述符中的地址 BASE+0 处，描述符存放于存储器中。

表 3.6　DMA 配置字中的各个位

位	名　　称	值
0	DMA 使能	0 禁止，1 使能
1	方向	取决于外设定义
2	完成中断使能	0 禁止，1 使能
3	数据尺度	取决于外设定义
4	自动缓存	取决于外设定义
5～6	控制状态	取决于外设定义
7	缓存清零	在配置字中必须清零
8	出错中断使能	取决于外设定义
9～11	状态	取决于外设定义
12～13	缓冲状态/数据长度	取决于外设定义
14	DMA 完成状态	0 成功，1 出错
15	描述符所有权	0 处理器所有，1 位 DMA 通道所有

注意，在某些外设中，DMA 配置字的第 12 位读时作为缓冲状态，写时作为数据长度。

每个 DMA 通道都有独立外设的一些功能。通过 DMA 配置字可以控制或监视独立外设的下列行为：

（1）数据传输方向（位 1），可读可写，对某些外设和 MemDMA 该位是不能修改的。

（2）数据长度（位 3 和位 12），配置方式参见表 3.7。传输 8 位数据时地址增量为 1，16 位时增量为 2，32 位时增量为 4。

表 3.7　DMA 配置字的数据长度设置

第 12 位	第 3 位	数 据 长 度
0	0	16 位半字
0	1	32 位字
1	0	保留
1	1	8 位字节

注意，在某些外设中，DMA 配置字的第 12 位读时作为缓冲状态，写时作为数据长度。

（3）外设控制（位 5 和位 6），独立外设控制位提供了通过描述符对外设的控制。

（4）外设状态（位 9～位 13），独立外设状态位含有与当前描述符 DMA 传输有关的包括缓冲器状态在内的外设说明信息。在 DMA 传输完成时，该信息被写回当前描述符的 DMA 配置字中。

基于描述符的 DMA 的设置的典型时序如下。在进行操作之前，必须将描述符基地址（BASE+0）的高 16 位设置到 DMA 描述符的基指针寄存器（DMADBP）中。

将 DMA 配置字（位 15 置 1）、DMA 传输计数、DMA 起始地址[15:0]和[31:16]，以及下一个描述符指针[15:0]写入描述符存储器地址 BASE+0～BASE+8 中。如果描述符是链表中的最后一个元素，或者是链表中的唯一元素，那么该描述符中的下一个描述符指针应该指向一个存储单元，该存储单元的位 15 必须置为 0。基于上述要求，下一个描述符指针可以指向当前描述符的基地址 BASE+0。在 DMA 传输序列完成之后，DMA 配置字的位 15 被清零，控制权返回给处理器。

将描述符基地址（BASE+0）的低 16 位写到相应外设的下一个描述符指针寄存器中。

设置相应外设的 DMA 配置寄存器中的 DMA 使能位。仅仅在单个 DMA 传输序列或者链表中的第一个 DMA 传输序列中，需要进行这一步操作。

一旦检测到（外设）DMA 配置寄存器中的 DMA 使能位有效，DMA 通道即读取描述符中的第一个元素——DMA 配置字，并将它复制到 DMA 配置字寄存器中。然后检测该寄存器的位 15，以确定该描述符是否被配置，是否做好被使用的准备。

如果位 15 为 0，则 DMA 通道被延迟并且等待，直到处理器对外设的 DMA 描述符准备寄存器进行写操作。这个写操作触发 DMA 控制器重新复制 DMA 配置字到 DMA 配置寄存器中，然后重新检测该寄存器的位 15。

如果位 15 为 1，则 DMA 通道读取描述符中剩余的 4 个元素，并把它们载入相应的 DMA 控制寄存器中，然后开始 DMA 传输。传输序列的状态在对应的 DMA 状态寄存器中每个周期更新一次。当 DMA 传输计数寄存器的值被减到 0 时，DMA 传输序列完成。传输过程完成后，DMA 通道将执行以下动作：

（1）清零 DMA 配置寄存器，将所有权返回给处理器；

（2）复制 DMA 配置寄存器的内容到当前描述符的 DMA 配置字中，该回写包括传输过程的最终状态；

（3）在允许中断的情况下，传输中断；

（4）读取下一个描述符的配置字，过程继续或终止。

2．基于自动缓冲的 DMA 传输

基于自动缓冲的 DMA 传输的操作，除了不需要存储器中的描述符外，其他的与基于描述符的 DMA 传输相同。在自动缓冲模式中，DMA 控制寄存器是可写的，并且可以被处理器直接编程以发起一个 DMA 传输过程。一旦传输过程完成，控制寄存器将被它们的原始设置值重载，以备下一次传输。这将产生一个循环缓冲区，并且不断地传输数据，直到将（外

设）DMA 配置寄存器的 DMA 使能位清零后才被禁止。如果允许中断，则在传输过程完成时将产生中断。

下面是设置基于自动缓冲 DMA 的典型步骤。

（1）将 DMA 配置器中的自动缓冲使能位置 1；

（2）初始化 DMA 传输计数器、DMA 起始地址高低位寄存器；

（3）写 DMA 配置寄存器，对 DMA 传输进行设置，将 DMA 使能位置 1。这一步写操作将启动基于自动缓冲的 DMA 的传输过程。

要停止 DMA 传输，需将配置寄存器中的 DMA 使能位清零。在传输期间将该位清零，不会像基于描述符的 DMA 那样，引起 DMA 错误结束。一旦检测到 DMA 使能位被清零，DMA 通道便结束当前单个或突发式传输，然后禁止 DMA。在该位清零后，DMA 起始地址和 DMA 传输寄存器可以用来监视 DMA 传输的状态。

3．存储器 DMA（MemDMA）

存储器 DMA 控制器提供在 Blackfkin 存储空间中的存储器对存储器的 DMA 传输。这些存储空间包括外设组件互联（PCI）地址空间、L1、L2 及外部的同步和异步存储器。

MemDMA 控制器由两个通道组成，一个是源通道，用于读存储器；另一个是目的通道，用于写寄存器。两个通道共享一个可存放 16 条记录的 32 位 FIFO。源 DMA 通道写 FIFO，目标 DMA 通道读 FIFO。FIFO 深度大大改善了内部和外部存储器之间的数据传输流量。FIFO 支持 8/16/32 位的传输。但是在 8/16 位传输中，仅仅使用 32 位数据总线的一部分，因此这两种传输的流量要比全 32 位 DMA 操作低。

MemDMA 控制器不支持自动缓冲的 DMA。使用 MemDMA 控制器的传送必须是基于描述符的 DMA。要有两个独立的描述符链表，一个用于源 DMA 通道，另一个用于目标 DMA 通道。这种分别控制方式，能使片外主处理器通过 PCI 接口总线管理其中一个表，而另一个表由内核 DSP 管理。因为源和目标 DMA 通道共享一个 FIFO 缓冲，两个描述符中的传输计数必须设置为相同的值，数据尺寸也设置为相同的值。

3.3.8　系统接口

Blackfin DSP 提供了多种外设接口，包括各种串口（同步外设接口 SPI、串行接口 SPORT 和统一异步收发器 UART）、USB 接口、PCI 接口和并口 PPI（支持视频输入和输出）。

1．串行接口（SPORT）

ADSP-BF533 处理器组合两个双通道同步串口（SPORT0 和 SPORT1）用于串行和多处理器通信。SPORT 支持以下特性：

（1）I^2S 能力操作。

（2）双向操作。每个 SPORT 有两组独立的传输/接收引脚，提供 8 条立体声音频通道。

（3）缓存（深度为 8）传输和接收端口。每个端口有一个数据寄存器，用于与其他处理器模块之间传输数据，同时对寄存器进行移位操作，以便将数据移入或移出数据寄存器。

（4）计时（时钟）。每个传输和接收端口或使用外部串行时钟，或生成自己的时钟——频率范围从（f_{SCLK}/131070）Hz 到（f_{SCLK}/2）Hz 之间。

（5）字长。每个 SPORT 支持从 3 位到 32 位的串行数据字长度，可以先传输最重要位，也可以先传输最不重要位。

（6）成帧。每个传输和接收端口对每个数据字的传输，可以采用帧同步信号，也可以不采用帧同步信号。帧同步信号可以从内部或外部产生。

（7）硬件压扩。每个 SPORT 可以按照 ITU G.711 标准执行 A 律或 μ 律压扩。压扩可以选择在 SPORT 的传输和/或接收通道，没有任何额外的响应延迟。

（8）单周期开销 DMA 操作。每个 SPORT 可以自动接收和发送多缓冲区的内存数据。处理器可以在 SPORT 和内存之间将一序列 DMA 传输链接或串联起来。

（9）中断。每个传输和接收端口在完成一个数据字的传输，或传输完整个数据缓冲区或多个数据缓冲区（通过 DMA）后产生中断。

（10）多通道能力。每个 SPORT 支持 1024 通道窗口中的 128 个通道，并且与 H.100、H.110、MVIP-90 及 HMVIP 标准是兼容的。

2．串行外设接口（SPI 端口）

ADSP-BF533 处理器有一个 SPI 兼容端口，可以是处理器与多个 SPI 兼容设备通信。SPI 接口使用 3 个引脚传输数据：两个数据引脚（主出从入 MOSI 和主入从出 MISO），一个时钟引脚（串行时钟 SCK）。SPI 芯片选择输入引脚（SPISS）让其他 SPI 设备选择处理器，7 个 SPI 芯片选择输出引脚（SPISEL7～1）让处理器选择其他 SPI 设备。SPI 选择引脚是重配置的通用 I/O 引脚。使用这些引脚，SPI 端口提供全双工、同步串行接口，能够支持主模式和从模式及多种环境。

SPI 端口的波特率和时钟相位/极性都是可编程的，它还有一个集成的 DMA 控制器，可以配置为支持传输或接收数据流。SPI 的 DMA 控制器在任何时刻只能服务于单方向访问。SPI 端口时钟速率计算如下

$$\text{SPI时钟速率} = \frac{f_{SCLK}}{2 \times \text{SPI\_BAUD}} \tag{3-1}$$

此处 16 位 SPI_BAUD 寄存器取值为 2～65535。

在传输中，SPI 端口同时传输和接收，通过串行的将数据移入或移出它的两个串行数据线。串行时钟线负责同步这两条数据线上的移位操作和数据采样操作。

3．统一异步收发端口（UART）

ADSP-BF533 处理器提供一个全双工通用异步接收/发送（UART）端口，它与 PC 标准的 UART 完全兼容。UART 端口提供到其他外设或主机的简化的 UART 接口，支持串行数

据的全双工、支持 DMA 的异步传输。UART 端口支持 5～8 个数据位，1 个或 2 个停止位，无奇偶校验或奇校验或偶校验。UART 端口支持以下两种模式的操作。

（1）指定的 I/O：处理器通过写或读 I/O 映射的 UART 寄存器来发送或接收数据。数据在发送和接收上采用双重缓存。

（2）DMA：DMA 控制器传输发送和接收数据。这减少了与内存之间发送和接收数据所需要中断的次数和频率。UART 有两个专用的 DMA 通道，一个用于发送，另一个用于接收。这些 DMA 通道比大多数 DMA 通道具有更低的优先级，因为它们具有较低的服务速率。

波特率、串行数据格式、错误码生成与状态、用于 UART 端口的中断等，都是可编程的。UART 可编程特征包括如下：

（1）支持的位率从（f_{SCLK}/1 048 576）bps 到（f_{SCLK}/16）bps。

（2）支持的数据格式每帧 7～12 位。

（3）发送和接收操作都可以配置为产生对处理器的可屏蔽中断。

UART 端口的时钟速率计算如下

$$\text{UART时钟速率} = \frac{f_{SCLK}}{16 \times \text{UART\_Divisor}} \tag{3-2}$$

此处 16 位 UART_Divisor 来自 UART_DLH 寄存器（最重要的 8 位）和 UART_DLL 寄存器（最不重要的 8 位）。

4．USB

Blackfin 中的通用串行设备（USBD）模块包括一个 USB 控制器内核（UDC）和一个前端接口。UDC 支持所有的 USB 协议，并且为前端提供了一个简单的读写接口。为了与 Blackfin 处理器的外设总线相连及支持一个操作系统，前端与 UDC 之间增加了一些硬件。

USBD 模块还包括存储器映射的寄存器、中断子系统、UDC 的配置模式和时钟控制，以及一个用于控制 USB 端点数据在系统内存和 USB 之间传输的 DMA 控制器。

在系统这一侧，USBD 模块连接数据访问总线（DAB）和外设访问总线（PAB）。在 USB 这一侧，USBD 模块连接片外的 USB 收发机。

Blackfin USB 外设功能包括如下：

（1）8 个物理 USB 端点。

（2）支持所有 USB 传输类型。

（3）具有低功耗能力，时钟可停止。

（4）与 Blackfin 中断系统通过单根中断线连接。

（5）作为从设备与 PAB 连接一般系统访问寄存器。

（6）作为主设备与 DAB 连接，可通过 DMA 主机方式访问 L2 存储器。

（7）主机总线仲裁机制在收到 DMA 传输请求的 3～5 个 USB 周期内，分配 USB DMA 通道。

5．PCI 接口

Blackfin 包含一个遵守 PCI 局部总线规范修订版 2.2 版的 33MHz 的 32 位 PCI 总线接口。PCI 接口在外部 PCI 总线和存储器、外设之间，外部 PCI 总线和处理器内核（此处理器内核位于 Blackfin 内部）之间提供总线桥接功能。PCI 总线接口支持以下功能：

（1）PCI 设备。所以 Blackfin 的系统可以很容易地与一个 3.3V、遵守 PCI 总线 2.2 版的 PCI 总线接口。

（2）主机到 PCI 总线的桥路。在桥路中，Blackfin 资源（处理器内核、内外部存储器及存储器 DMA 控制器）提供必需的硬件作为 PCI 总线的系统控制器。从一个或多个 PCI 设备来讲，这为 Blackfin 提供了外设扩展。

在这两种设置中，PCI 可以在 PCI 总线会话中用作 PCI 发起者（主设备）或 PCI 目标设备（从设备）。

6．并行外设接口（PPI）

处理器通过 PPI 可以直接与平行的 ADC 和 DAC、视频编解码器及其他通用外设连接。PPI 由专用的输入时钟引脚、最多 3 个帧同步引脚和最多 16 个数据引脚组成。输入时钟支持最高半系统时钟速率的数据速率，同步信号可以配置为输入或输出。

PPI 支持多种通用的和 ITU-R 656 模式的操作。在通用模式下，PPI 提供半双工、双向数据传输，最高为 16 位的数据。还提供了最多 3 个帧同步信号。在 ITU-R 656 模式，PPI 提供 8 位或 10 位视频数据的半双工双向传输。另外，还支持片上对嵌入式 SOL（行起始）和 SOF（场起始）前对齐包的解码。

PPI 的通用模式是为了适应大量各种数据捕获和传输应用。支持 3 个不同的子模式：

- 输入模式：帧同步和数据是 PPI 的输入；
- 帧捕获模式：帧同步是 PPI 的输出，数据是 PPI 的输入；
- 输出模式：帧同步和数据是 PPI 的输出。

（1）输入模式：用于 ADC 应用，以及与硬件信号的视频通信。在它最简单的形式中，PPI_FS1 是一个外部帧同步输入，它控制何时去读数据。PPI_DELAY MMR 允许在收到该帧同步和数据读初始化之间有一个延迟（几个 PPI_CLK 周期）。输入数据样本数目是用户可编程的，由 PPI_COUNT 检测的内容定义。PPI 支持 8 位和 10 位直到 16 位数据，可在 PPI_CONTROL 寄存器中编程设定。

（2）帧捕获模式：允许视频源作为从属的（如用于帧捕获）。处理器控制何时从视频源读取。PPI_FS1 是一个 HSYNC 输出，PPI_FS2 是一个 VSYNC 输出。

（3）输出模式：用于发送视频或其他数据，以及最多 3 个帧同步信号。典型的，单个帧同步对于数据转换应用是合适的，2 个或 3 个帧同步对于发送带有硬件信号的视频是合适的。

ITU-R 656 模式是为了适合大量各种视频捕获、处理和传输的应用。它支持 3 个不同的

子模式：仅主动视频模式、仅垂直消隐模式和整场模式。

（1）仅主动视频模式：指只对一场中视频的活动部分感兴趣，对消隐等不感兴趣。PPI 不会读取 EAV（活动视频结束）和 SAV（活动视频开始）前同步之间的任何数据，也不读取垂直消隐段的任何数据。在该模式中，控制字节序列不存到内存，它们被 PPI 滤除了。与场 1 同步后，PPI 忽略 SAV 码之前的样本。用户指定每帧（在 PPI_COUNT 寄存器中）主动视频行数。

（2）仅垂直消隐间隔模式：PPI 只传输垂直消隐间隔（VBI）数据。

（3）全场模式：通过 PPI 读取所有输入的位流。这包括了主动视频、控制前同步序列及可以嵌入垂直和水平消隐间隔的辅助数据。与场 1 同步后数据传输马上开始。数据通过 8 个 DMA 引擎（与处理器内核是自治的）与同步通道进行数据传输。

7．指令系统

Blackfin 处理器家族汇编语言指令集合利用代数语法，易于编码并提高了可读性。这些指令经过特定的调整，能够提供灵活的、高密度编码的指令集合，最终编译结果占用内存非常小。该指令集合还提供了全特征的多功能指令，允许程序员在单个指令中使用多个处理器内核资源。与许多微控制器中常见的特征相结合，该指令集合在编译 C 和 C++源代码时非常高效。另外，该结构支持用户模式（算法/应用代码）操作和超级用户模式（OS 内核、设备驱动器、调试器、ISR 等）操作，允许对核心处理器资源的多级别访问。

汇编语言利用了处理器的独特结构，具有以下优点：

（1）对 8 位和 16 位操作，无缝集成的 DSP/CPU 特性达到最优化。

（2）多发载入/存储的改进 Harvard 结构，每个周期支持两个 16 位 MAC 或 4 个 8 位 ALU，加上两个载入/存储，再加上两个指针更新。

（3）所有寄存器、I/O 和内存都映射到一个统一的 4G 字节内存空间，提供了简单而统一的编程模型。

（4）微控制器特性，如任意的位和位字段操作、插入和抽取；对 8/16/32 位数据类型的整型操作；单独的用户和超级用户堆栈指针。

（5）代码密度增强，包括使 16 位和 32 位指令混合（不存在模式切换，不存在代码隔离）。经常使用的指令以 16 位进行编码。

3.4　ADSP 开发过程

ADSP 软件开发流程包括结构描述、源代码生成、软件验证（调试）及硬件实现等步骤。图 3.10 显示了一个典型的开发周期。

结构描述：用户首先创建一个关于目标硬件系统的软件描述。系统描述文件包括所有可用内存、内存映射的外围设备等。

源代码生成：该步骤将算法思想转化为运行于 DSP 上的代码。有些程序员喜欢用高级

语言（如 C）写代码，也有的喜欢使用处理器汇编语言。用 C 开发速度快，但是编译出来的 DSP 代码缺乏效率，因为没有充分利用处理器结构的优势。汇编代码充分利用处理器的优势，产生高效的实现。但是需要对处理器汇编语言很熟悉。最有效的是混合 C 和汇编代码（对于时间敏感、数学计算量大的部分）两种模式。任何情况下，程序员必须了解处理器限制和外设性能指标。

软件验证（调试）：利用软件工具（仿真器）测试生成代码的结果，检查程序的逻辑流，验证算法符合设计目的。仿真器是 DSP 处理器的一个模型：提供所有内存和处理器寄存器的可见性；允许用户连续或单步运行 DSP 代码；可以仿真外部设备向处理器馈入数据。

硬件实现：代码运行在真实的 DSP 上。典型的分为几个阶段：在评测平台（如 EZ-KIT Lite）上尝试；在运行电路内仿真；产品 ROM 生成。评测可以快速决定是否进行下去。运行电路内仿真在系统中监视软件调试，可以利用 EZ-ICE 等工具控制目标平台上的处理器操作。当所有调试完成，生成最终代码的引导 ROM，作为最终的产品实现。

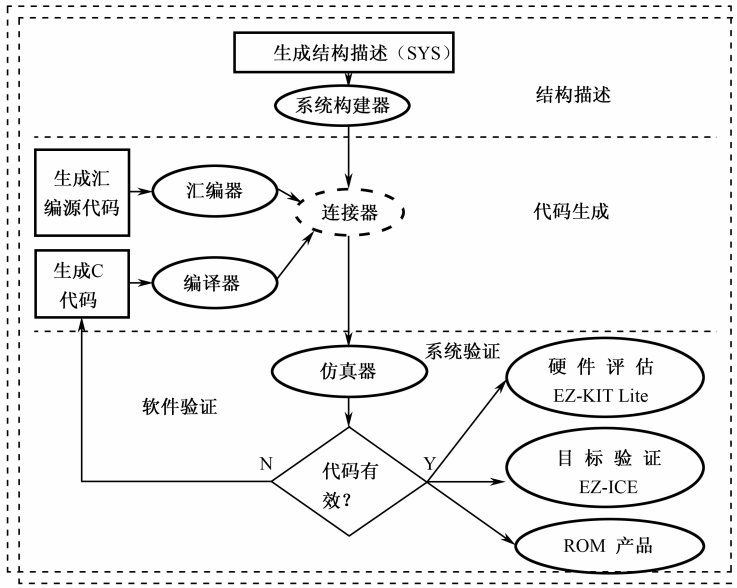


图 3.10　DSP 软件开发流程

下面将以 FIR 滤波器为例介绍 DSP 系统的开发过程。

数字化地实现该滤波器需要两步。

（1）将滤波器公式看做计算机程序，将公式分解为数学步骤（如乘法和加法），并确定软件实现所需的所有额外操作，如处理指令和数据、测试状态等。

（2）将这些操作写成程序，这可能比较费力。这可以用高级语言（如 C 语言），也可以用汇编语言。高级语言编码效率更高，但系统性能需要优化时，汇编语言会更有用。

图 3.11 中给出了滤波器方程式及其延迟线模型，接下来给出用 C 语言实现 FIR 算法的代码。

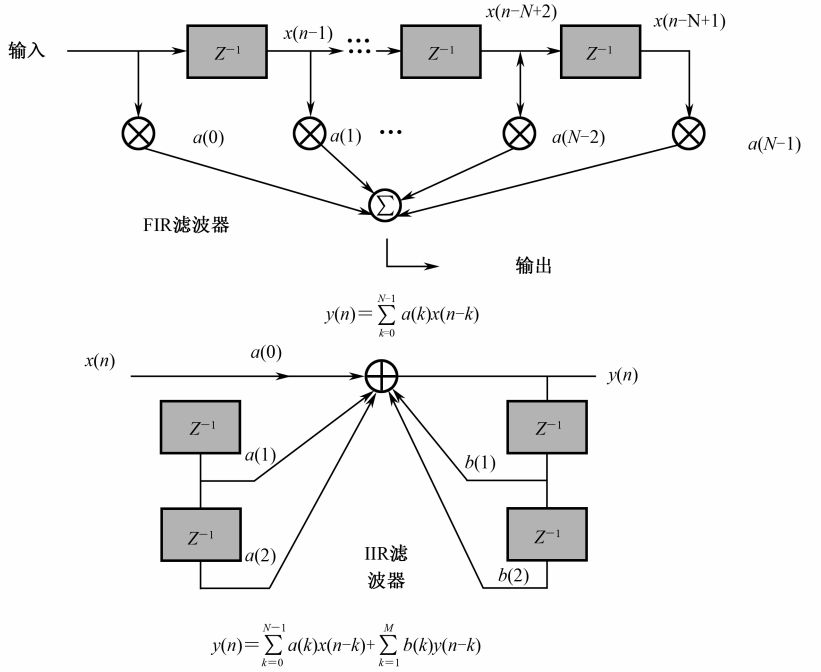


图 3.11　滤波器方程式及其延迟线模型

```
float fir_filter(float input, float *coef, int n, float *history) {
    int i;
    float *hist_ptr, *hist1_ptr, *coef_ptr, output;
    hist_ptr = history;
    hist1_ptr = hist_ptr;     /* 用于历史更新 */
    coef_ptr = coef + n -1;   /* 指向最后的系数 */
    /* 形成输出累积 */
    output = *hist_ptr++ * (*coef_ptr-);
    for(i = 2; i < n; i++) {
        *hist1_ptr++ = *hist_ptr;      /* 更新历史组数  */
        output += (*hist_ptr++) * (*coef_ptr-);
    }
    output += input * (*coef_ptr);
    *hist1_ptr = input;
    return(output);
}
```

下面给出的是利用线性汇编语言实现的 FIR 滤波器函数。其中利用了循环缓冲区、零开销硬循环、单指令多任务（SIMD）等 DSP 的特性，能够极大提高 FIR 滤波器的计算效率。

```
.section program;
.global    __fir;
```

```
.align 8;
__fir :
 P0=[SP+12];          // 滤波器结构地址
 nop;nop;nop;
 P1=[P0++];           // 滤波器参数数组地址
 P2=[P0++];           // 延迟线地址
 R3=[P0++];           // 滤波器参数个数
 B3=R1;               // 输出缓冲区初始化为循环缓冲区
 I2=P1;               // I2 初始化为参数数字起始地址
 B2=P1;               // 滤波器参数数组初始化为循环缓冲区
 I0=P2;               // 延迟线写起始地址
 B0=P2;               // 延迟线缓冲区初始化为循环缓冲区
 I1=P2;               // 延迟线读起始地址
 B1=P2;               // 延迟线缓冲区初始化为循环缓冲区
 I3=R1;
 P1=R2;
 P2=R3;
 R2=R2+R2;
 CC=BITTST(R3,0);     // 检查滤波器拍数是否为奇数
 R3=R3+R3;            // 因为滤波器参数好似 fract16，占用 2 字节
 L2=R3;               // 初始化滤波器参数长度寄存器
 P0=R0;               // 输入数组地址
 IF !CC JUMP FIR_CONTINUE (BP);
 R3+=2;               // 使滤波器拍数为偶数
 L2=R3;
 NOP;NOP;NOP;NOP;
 I2-=2;               // 参数要补零的地址
 R0=0;
 W[I2++]=R0.L;        // 令最后一个滤波器参数为 0，使滤波器拍数为偶
FIR_CONTINUE:
 L0=R3;               // 设置延迟线缓冲区长度
 L1=R3;               // 设置延迟线缓冲区长度
 L3=R2;
 P2+=-1;              // Nc-1（滤波器参数 - 1）
 nop;nop;nop;
 I3-=4;               // 调整输出真正到最后到最后的位置上
/*****************************************************************/
NOP ;                          // 对齐指令
 I0+=4 || R2.H=W[I2--];        // 调整延迟写指针为 X-n，参数读指针为 h-n
```

```
    R2.H=W[I2++] || R1=[I0];      // 取 hn-1，取 X-n 和 X-n+1
     R0=[P0++];                   // 取 X0 和 X1
     // 建立零开销循环（外层）
     LSETUP(E_FIR_START,E_FIR_END) LC0=P1>>1; // 从 1 到 Ni/2 循环
    E_FIR_START:
     R1=PACK(R1.H,R0.H) || [I0++]=R0 || R2.L=W[I2++];
                                  // 把 X1 保存到 R1 的低半部分
                                  // 更新延迟线
                                  // 将 h0 取到 R2 的低半部分
     // 建立零开销循环（内层）
     LSETUP(E_MAC_ST,E_MAC_END)LC1=P2>>1; // 从 1 到 Nc/2 - 1 循环
         A1=R2.L*R1.L, A0=R2.H*R1.H || R2.H=W[I2++] || [I3++]=R3;
                                  // A1=h0*X1, A0=hn-1*X-n+1.
                                  // 取 h1 到 R2 的高半部分；保存输出结果
    E_MAC_ST:     // 以下三行代码可以并行执行
                  A1+=R0.L*R2.H,A0+=R0.L*R2.L || R2.L=W[I2++] || R0=[I1--];
                  // A1+=R0.L*R2.H,A0+=R0.L*R2.L;
                  // R2.L=W[I2++]; //A1+=X0*h1, A0+=X0*h0
                  // R0=[I1--];         //将滤波器参数 h2 到 R2 的低半部分。
                                        //取 X-1 和 X-2 到 R0 的上半部分和下半部分
    E_MAC_END:    A1+=R0.H*R2.L,A0+=R0.H*R2.H  || R2.H=W[I2++] ;
                                        //A1+=X-1*h2, A0+=X-1*h1
                                        //取 h3 到 R2 的上半部分
    E_FIR_END: R3.H=(A1+=R0.L*R2.H), R3.L=(A0+=R0.L*R2.L)  || R0=[P0++] ||
R1=[I0];
                // A1+=X-n+2*hn-1, A0+=X-n+2*hn+2
                // 取下一对输入（X2 和 X3）到 R0 的下半部分和上半部分
                // 取 X-n+2 和 X-n+3 到 R1 中
     [I3++]=R3;           // 保存最终的滤波器输出
    /*****************************************************************/
         L0=0;           // 清除循环缓冲区初始化
     L1=0;
     L2=0;
     L3=0;
     RTS;
    __fir.end:
```

该汇编程序实现的函数可以很方便地由 C 代码来调用，对应的数据结构定义和调用语句如下：

```
    typedef struct {
```

```
    fract16 *h;        /*  滤波器系数      */
      fract16 *d;      /*  延迟线起始地址  */
      int k;           /*  系数个数        */
} fir_state_fr16;
int nsamples;
fract16 OUT[BUFFER_SIZE];
fract16 IN[BUFFER_SIZE];
_fir(IN, OUT, nsamples, &s);
```

为了响应外部事件，如等待收到 ADC 的中断后立即调用 FIR 函数，完整的程序还需要包括中断向量表更新、中断寄存器设置等初始化过程。初始化后，DSP 执行主程序的指令，执行背景任务，循环处理代码，或以低功率备用模式空转，直到从 ADC 获得中断后立即执行 FIR 程序，从内存读取数据和滤波器参数，处理输入样本后，将结果输出到指定位置。

为得到可执行文件，首先对 DSP 代码进行汇编，汇编的同时检查代码的语法错误。然后将代码链接起来，利用在结构文件中声明的可利用的内存输出 DSP 可执行文件。连接器将所有代码和数据从源代码安置到内存空间，输出 DSP 可执行文件，它可以下载到 EZ-Kit Lite 板上。

利用 EZ-Kit LITE 板开发，代码需要合并一些 EZ-KIT Lite 的特性，比如说 CODEC 初始化和数据 I/O。而滤波器算法的核心元素则没有变化。

在算法代码方面，为了实现该滤波器需要使用 MAC 计算单元和数据地址生成器。FIR 滤波器需要在每个数据点、每一拍上重复乘—累积操作。为此 MAC 指令采用循环的形式。ADSP 的零开销循环功能允许 MAC 指令重复指定次数而无需软件介入。计数设置为拍数减 1，循环机制每次循环自动对计数减 1。设置循环计数为拍数确保了完成处理后数据指针在正确的地方停止，并且允许最终的 MAC 操作包括四舍五入。

为最优化代码执行，每个指令周期应该执行一个有意义的数学计算。ADSP 利用多功能指令实现这个目的：处理器可以在一个指令周期执行几个功能。对于 FIR 滤波器代码，每个 MAC 操作可以并行执行：两个数据访问，一个来自数据内存，一个来自程序内存。该功能意味着在每个循环迭代可以执行一个 MAC 操作。同时，下一个数据值和参数被获取，计数自动减 1。所有这些都没有浪费时间来维护循环。

下面的代码显示了在 EZ-KIT Lite 平台上通过 AD1836 采集音频信号的关键代码片段。从 main 函数可以看出，实现要完成对系统的初始化，包括对 EBIU、Flash、AD1836、SPORT0、DMA 的初始化；然后要设置相关的中断处理寄存器，中断处理函数 ISR 中则是要进行相关算法（如 FIR 滤波）的地方。

```
void main(void) {
 sysreg_write(reg_SYSCFG, 0x32);     // 初始化系统配置寄存器
 Init_EBIU();
 Init_Flash();
```

```
 Init1836();
 Init_Sport0();
 Init_DMA();
 Init_Interrupts();
 Enable_DMA_Sport0();
 while(1);                                    // 等待中断事件的发生
}
```

部分初始化函数如下：

```
void Init_EBIU(void) {
 *pEBIU_AMBCTL0  = 0x7bb07bb0;
 *pEBIU_AMBCTL1  = 0x7bb07bb0;
 *pEBIU_AMGCTL   = 0x000f;
}
…
void Init_Sport0(void) {
 // SPORT0 接收：外时钟，外帧同步，高位先，低有效，24 位，
 //              立体帧同步
 *pSPORT0_RCR1 = RFSR | RCKFE;
 *pSPORT0_RCR2 = SLEN_24 | RXSE | RSFSE;
 // SPORT 传输：外时钟，外帧同步，高位先，低有效，24 位
 // 辅助声道启动，立体声帧同步
 *pSPORT0_TCR1 = TFSR | TCKFE;
 *pSPORT0_TCR2 = SLEN_24 | TXSE | TSFSE;
}
void Enable_DMA_Sport0(void) {
 // 使能 DMA
 *pDMA2_CONFIG   = (*pDMA2_CONFIG | DMAEN);
 *pDMA1_CONFIG   = (*pDMA1_CONFIG | DMAEN);
 // 使能 Sport0 TX 和 RX
 *pSPORT0_TCR1   = (*pSPORT0_TCR1 | TSPEN);
 *pSPORT0_RCR1   = (*pSPORT0_RCR1 | RSPEN);
}
void Init_Interrupts(void) { // 初始化中断服务
 // 设置 Sport0 RX (DMA1)中断优先级为 2= IVG9
 *pSIC_IAR0 = 0xffffffff;
 *pSIC_IAR1 = 0xffffff2f;
 *pSIC_IAR2 = 0xffffffff;
 // 为中断向量 SPORT0 RX 配置 ISR -> IVG 9
```

```
    register_handler(ik_ivg9, Sport0_RX_ISR);
    // 使能 Sport0 RX 中断（接触屏蔽）
    *pSIC_IMASK = 0x00000200;
}
```

定义对应的服务器程序 ISR：

```
EX_INTERRUPT_HANDLER(Sport0_RX_ISR) // 定义 ISR
{
    // 确认中断处理
    *pDMA1_IRQ_STATUS = 0x0001;
    // 将输入数据从 DMA 输入缓冲区复制到变量中
    iChannel0LeftIn = iRxBuffer1[INTERNAL_ADC_L0];
    iChannel0RightIn = iRxBuffer1[INTERNAL_ADC_R0];
    iChannel1LeftIn = iRxBuffer1[INTERNAL_ADC_L1];
    iChannel1RightIn = iRxBuffer1[INTERNAL_ADC_R1];
    // 调用用户的数据处理函数
    Process_Data();
    // 将处理过的数据从变量复制到 DMA 输出缓冲区中
    iTxBuffer1[INTERNAL_DAC_L0] = iChannel0LeftOut;
    iTxBuffer1[INTERNAL_DAC_R0] = iChannel0RightOut;
    iTxBuffer1[INTERNAL_DAC_L1] = iChannel1LeftOut;
    iTxBuffer1[INTERNAL_DAC_R1] = iChannel1RightOut;
}
```

3.5 集成开发套件 VisualDSP++简介[31,32]

3.5.1 开发工具及其特点

VisualDSP++是针对 ADI 公司 DSP 器件专门开发的开发平台，支持 ADI 公司所有系列的 DSP 处理器。VisualDSP++集成了两大部分：集成的开发环境和调试器。VisualDSP++具有灵活的管理体系，为处理器应用程序和项目的开发提供了一整套工具。VisualDSP++包含生成和管理处理器项目必须的所有工具。

（1）与 VisualDSP++一体化的集成开发和调试环境（IDDE）。

（2）带有实时运行库的 C/C++语言最优化编译器。

（3）汇编程序、连接器、预处理器和档案库。

（4）程序加载器、分割器。

（5）模拟器。

（6）EZ-KIT Lite 评估系统（需要单独购买）。

（7）仿真器（需要单独购买）。

（8）程序实例。

VisualDSP++有以下基本特点。

1．源文件编辑特点

VisualDSP++简化了源文件的操作，很容易实现创建、查看、打印、移动和信息定位等相关文件操作。源代码文件是 DSP 工程的重要组成，可采用 C/C++或汇编语言编写。如果采用汇编语言编写，DSP 工程中还应包含链接描述文件（.LDF 文件）和相关数据文件。编辑窗口即 VisualDSP++编辑器，是一个完整的代码书写工具，用于编辑文本文件。其中，为专用语法配置颜色，提高查看和搜索效率，利用状态图来标明断点、书签和处理器当前指令对应的源代码位置。还能够查看错误信息和违规代码，可在输出窗口看到错误的详细信息，双击错误行可跳至编辑窗口中的违规代码处。

2．工程管理特点

工程管理包括创建、定义和编译等处理器项目所必需的操作。

定义和管理工程：管理工程编译时所需相关文件和开发工具。只需一次即可，在开发过程中可根据需要对工程灵活地进行修改。

查看和管理代码开发工具：确定代码开发工具如何处理输入文件和生成输出文件。工程配置选项可在工程定义时设置，也可在工程开发过程中修改。

查看工程编译结果：工程编译过程中可随时查看编译状态。如果存在错误，在输出窗口中双击错误信息即可查看造成错误的源代码。

管理源代码文件：根据源文件的依赖项，可显示文件之间的关系

VisualDSP++使用代码开发工具处理工程和生成处理器所需的程序。它还提供源代码控制（SCC）界面，用户可以直接在 IDDE 环境下完成源代码的控制。

3．调试特点

在调试工程的过程中，VisualDSP++提供以下一些工具和功能为用户服务。

（1）查看 C/C++和汇编语言联合编程的源代码文件。

（2）运行命令行脚本。使用脚本可以指定调试过程中的主要参数和特性。

（3）使用存储器相关的表达式。

（4）利用断点查看寄存器和存储器。可以快速添加、移除、使能、禁止断点。

（5）设置模拟观察点。对堆栈、寄存器、存储器等设置观察点可停止程序执行，方便调试过程中观察相关信息。

（6）统计目标处理器的指令执行数（仅用于 JTAG 仿真调试对象），图形化显示统计结果，可轻松观察到程序中最耗时的指令部分。

（7）线性描述目标处理器指令执行数（仅用于模拟调试对象）。对 DSP 的 PC 寄存器进行取样，统计其执行情况，并将结果用图形显示。

（8）模拟 I/O 端口数据流、中断的产生。

（9）创建用户自定义寄存器窗口。配置自定义寄存器窗口来显示指定寄存器组。

（10）根据存储器中的数值绘图。将存储器中的数据以图像形式显示出来，可以选择多种绘图风格、典型数据处理功能和外观显示方式。

（11）跟踪程序运行历史，可获得用户程序是如何达到用户设置的特定的程序点，并显示读、写和符号名称等相关信息。

（12）查看汇编指令流水线深度。通过流水线界面可查看处理器流水线的阶段。

4．VDK 特点

VDK 即 VisualDSP++内核，是一种可扩展的软件执行程序，集成于 VisualDSP ++中。VDK 方便用户从软件中访问硬件信息，使用户专心完成算法的实现。

VDK 为处理器应用开发过程提供了一组基本模块，性能描述如下。

（1）自动化。VisualDSP++可以根据用户指定的语言自动生成源代码框架。

（2）确定性。VisualDSP++明确指明 VDK 应用程序接口执行时间的确定性。

（3）多任务处理。VDK 任务（线程）之间相互独立，每个线程都有自己的栈。

（4）模块化。VDK 包含各种组件，在以后的版本中将会提供更多的功能。

（5）方便移植性。大部分核组件可用 ANSI 标准的 C/C++语言编写。

（6）优先性。VDK 的优先级调度表可使高优先级线程无需等待信号运行。

（7）原型化。VDK 和 VisualDSP++包含模板文件，方便用户创建原始文件，且整个应用程序是原型化的，需要用户根据需要进行测试和修改。

（8）可靠性。VDK 提供实时运行过程中的错误检查。

（9）可扩展性。如果某项目不包括该属性，则目标系统中不包含相关代码支持。

3.5.2　利用 IDDE 进行 DSP 程序开发

VisualDSP++的 IDDE 主要由工程管理窗口、文本编辑窗口、反汇编窗口、输出窗口和一些辅助菜单组成，如图 3.12 所示。IDDE 提供对 DSP 应用程序开发全过程的支持，一般都要经过如下几步。

（1）创建一个新的工程。

（2）设置工程选项。

（3）编辑或添加工程源文件。

（4）设置工程编译链接选项。

（5）编译链接 Debug 版的工程，生成可执行文件。

（6）建立 Debug Session 和加载可执行文件。

（7）运行和调试（Debug）程序。

（8）编译链接加载（Release）版本的工程。

通过以上 8 步即可方便地完成整个 DSP 的应用开发，下面分别介绍。

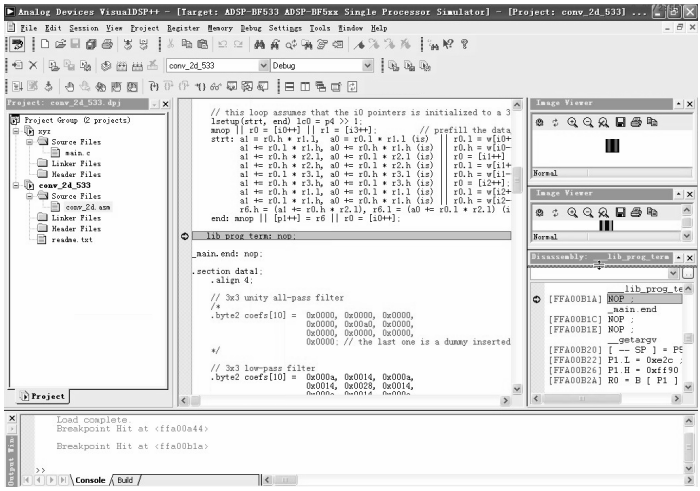


图 3.12　VisualDSP++ IDDE 用户界面

1．创建一个新的工程文件

VisualDSP++中 DSP 应用开发都是基于工程的。工程文件（*.Dpj）存放程序的编译链接信息：源文件列表、其关联关系信息和开发工具的选项设置等。要创建 DSP 工程，选中菜单项“File | New | Project”，启动新建工程向导，如图 3.13 所示。

此处工程名和路径由用户自己输入；工程类型由用户选择，包括标准应用、运行库、VDK 应用等，一般默认为标准应用。单击“Next”按钮，继续设置处理器类型、芯片版本号。处理器类型按需要选择 ADSP-BF561 或其他处理器。芯片版本号随所选处理器型号而变化，用户按需选择，未知的情况下可选自动或任意。

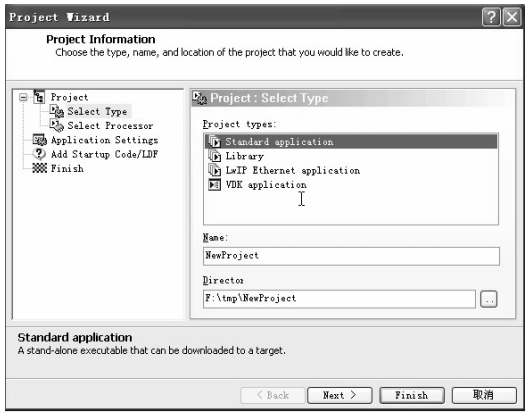


图 3.13　新建工程向导界面

接下来，在应用配置部分，选择是否添加模板源代码（C/C++或汇编），同时选择输出文件类型。输出文件类型包括可执行文件（.dxe）和加载文件（.ldr）。调试阶段一般设置为可执行文件，以方便模拟器或仿真器调试。开发完成阶段一般设置成加载文件，可以用于

对处理器进行程序加载。

对于 BF561 等双核芯片，接下来还需要设置双核属性：单核单应用、双核单应用、每核一应用及单应用使用双核。

接下来选择是否添加 LDF 和启动代码。至此工程配置工作完成，单击“Finish”按钮即可完成工程的新建工作，并在工程管理窗口中显示新工程。工程下通常有 3 个文件夹，分别对应源文件、链接文件和头文件。

2．设置工程选项

工程建立后，可通过工程选项窗口对工程修改，并设置工程参数。选中菜单项“Project | Project Options”显示出工程选项窗口，如图 3.14 所示。

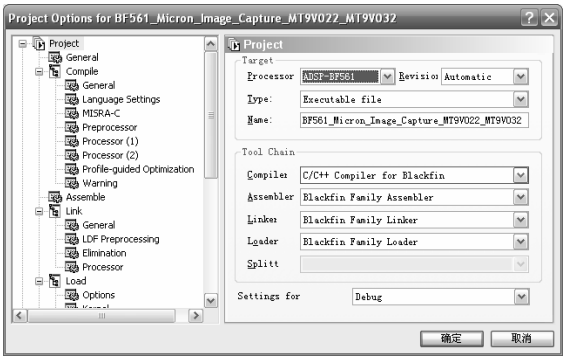


图 3.14　工程选项窗口

选项包含项目、常规、编译、汇编、链接、加载、预编译、后编译、LDF 设置、启动代码设置等部分。大部分选项可使用默认值。

项目（Project）栏用于选择处理器类型和工程输出类型，各选项意义如下。

目标（Target）：包括处理器类型、芯片版本、输出文件类型和输出文件名。

工具链组（Tool Chain）：指定编译器、汇编器、链接器、加载器和分割器。该组参数基本上使用默认值即可。

设置（Settings）：输出类型，为 Debug 或 Release。一般在调试过程中选择 Debug 类型，当程序调试好以后，选择 Release 类型。

3．编辑或添加工程源代码文件

工程一般包含多个 C/C++或汇编源代码文件。当工程进行编译链接时，IDDE 能自动选择可识别的文件进行编译链接。创建工程后，就可以编辑新的源代码文件或将已存在的源文件加入到该工程中。

添加文件到工程中一般可以采用 3 种方法：单击工具栏中添加文件图标；选中菜单项“Project | Add to Project | File(s)…”；在工程管理窗口中，选中需添加文件的工程，单击鼠标右键，选择菜单项“Add File(S)to Folder…”。弹出文件选择窗口后，查找并选中所需源文件

即可。被添加的文件会自动出现在工程管理窗口的文件目录列表中。

要新建一个文件并加入到工程中，选中菜单项“File | New”，或单击工具栏中新文件按钮，就会打开一个新编辑窗口。编辑器可以编辑任意名称的文本文件，文本编辑器将根据文件后缀名来判断文件类型，并根据文件类型以不同的颜色显示源代码文件中的关键字。编辑完成后，要把新编辑的文件存盘，并加入到工程中去。

相关性用于描述工程中源文件之间的相互关系（某个文件需要用到哪些文件的信息），它存放在后缀为.mak 的文件中，这决定了编译链接的顺序。更新工程相关性可以通过“Project | Update Dependencies”来实现。

4．设置工程配置选项

工程类型决定工程编译链接后的类型，可选择 Debug 或 Release。选择 Debug 并且接受其他默认值，编译器会产生一个包含调试信息的目标文件，供调试使用。选择 Release 并接受其他默认值，编译器会产生一个不包含调试信息的目标文件，并且会对代码进行优化。

5．编译链接 Debug 版的工程生成可执行文件

对工程配置完成后，使用 Build 方式对工程和相关文件进行编译和链接。可以通过工具栏上的图标或键盘快捷键 F7 来完成。也可以通过菜单“Project | Build Project”来完成。或者在工程管理窗口中，选择相应的工程，单击鼠标右键，在弹出菜单中使用“Bulid Project”选项来实现。

编译链接过程中，输出窗口中会显示状态信息。如果编译链接错误，输出窗口将会告知“编译链接失败”，且显示错误信息和错误类型。用鼠标双击出错信息行，IDDE 会自动打开出错的源代码文件，并跳转到与错误信息相关的代码位置。如果编译链接过程成功，输出窗口将显示“编译完成”等相关信息。

值得注意的是，输出文件必须指定为可执行类型，且工程类型为 Debug 时，才能产生可进行 Debug 调试的输出文件。

6．建立调试会话和加载可执行文件

编译链接成功后，VisualDSP++将生成处理器可执行文件。该文件可在 VisualDSP++自带的模拟器环境下运行，也可由计算机通过仿真器加载到目标板处理器中执行。二者都需要建立调试会话，建立会话完成后，由加载器将可执行文件调入模拟器或通过仿真器加载给目标板处理器执行。会话的相关操作请参见后面相关章节。加载完成后，将在输出窗口显示加载完成信息。

7．运行和调试程序

在完成可执行文件加载工作后，即可用 Debugger 工具来调试该工程了。通过单击工具栏上的图标或选择 Debug 菜单中的子菜单，就可以对程序进行运行、停止等调试操作。用户根据需要不断对程序进行修改、完善和优化。

如果工程不是最新的（有过期源文件或依赖信息），IDDE 会建议先编译链接此工程，然后再启动 Debugger 工具。

8．编译链接 Release 版的程序和生成加载文件

用户在对调试版的程序调试完成后，就基本完成了对 DSP 应用程序的开发。接下来可将 Debug 版程序优化后生成正式版本程序。如果该工程需要应用在硬件平台上，用户还需将正式版程序编译生成处理器加载文件，提供给处理器系统中的程序加载方法，实现处理器系统运行程序的加载。

生成正式版程序和加载文件可以通过以下简单的步骤完成。

（1）选择“Project| Project Configurations”，设置成“Release”；或选择“Project | Options”，将其中 Setting for 的内容修改为“Release”。

（2）选择“Project | Options”，将其中 Type 的内容修改为“Load File”。在 Options 窗口中，选择“Load”选项，根据用户系统的要求对所生成的加载文件进行设置

（3）编译链接该工程，VisualDSP++将生成正式版本的程序。

经过前面 8 个步骤，可以认为 DSP 处理器程序开发完成。

3.5.3 调试工具

1．设置调试会话

IDDE 中集成了 Debugger 工具，Debugger 可以直接应用 ADI 公司的模拟器（Simulator）和仿真器（Emulator）工具。

进行 Debugger 调试的第一步是设置调试会话（Debugging Sessions）。调试会话中主要是设置调试的目标和调试所使用的工具。Debugger 工具支持的会话类型包括硬件仿真调试会话和软件模拟调试会话。硬件调试会话必须有硬件系统的支持，而软件调试会话不需要硬件系统支持，是由 VisaulDSP++自带软件模拟器工具通过计算来模拟处理器的工作的。

新建调试会话的设置步骤如下：选中“Session | New Session”，弹出新建会话向导，如图 3.15 所示。新建会话向导的 3 个步骤分别是：处理器选择、连接类型选择和平台选择。

处理器选择用于设置所建立的调试会话是针对何种处理器、处理器型号的选择等。其中 Processor 栏用于选择目标处理器类型，此处选“Blackfin”。Choose a target processor 栏用于选择具体的处理器型号，如“ADSP-BF561”。

接下来进入连接类型选择窗口，用于设置所建立的会话使用的连接方式。一般有 4 种连接方式：评估板系统、仿真器、模拟器和遗留目标。用户根据需要自行选择其中的一种即可。模拟器方式不需要硬件支持，而评估板系统和仿真器连接方式需要用户提供硬件平台。如果用户没有硬件平台，而 VisualDSP++检测不到硬件设备，将会弹出错误信息，提示用户尝试连接硬件平台失败。

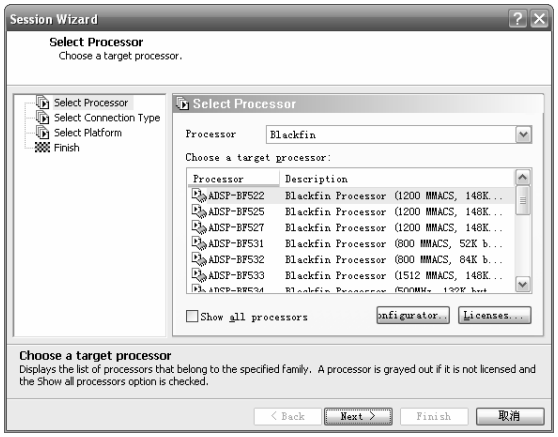


图 3.15　新建会话向导

下面将显示所建立的会话使用到的平台。对大多数处理器而言，该窗口中只有单一的选项。只有针对部分 Blackfin 系列处理器建立的平台才有两个选项，因此对于大多数情况，该窗口使用默认选项即可。

用户在确定所建立的会话信息无误后，单击完成后，VisualDSP++将根据用户的设置建立会话。会话建立完成后，VisualDSP++将把刚建立完成的会话作为活动会话。

如果用户已建立过多个会话，可以对存在的所有会话进行管理并在其中进行选择。通过菜单项“Session | Select Session”即可，选择已经存在的会话。用户还可以通过会话列表完成新建、删除和激活不同的会话。

2．程序执行操作

Debugger 中的程序执行命令在 Debug 下拉菜单中，在工具栏中也有相应的快捷按钮。下面简单说明一些常用的执行命令。

（1）运行（Run）。运行程序直到遇到某种条件（如断点、用户干预）才停止。程序处于停止状态时，IDDE 中所有已打开窗口的内容都更新为程序运行后的值。

（2）暂停（Halt）。程序执行过程中用户可暂停程序的执行。程序暂停状态时，IDDE 中所有已打开窗口的内容都更新为当前值，状态条显示当前程序停止的地址。

（3）执行到光标所在位置（Run To Cursor）。使程序运行，直到程序执行到光标所在位置时暂停。光标位置可在源文件窗口或反汇编窗口中设置。

（4）执行一行程序（Step over）。仅仅执行一行 C 语言程序行，仅用于 C 语言程序。

（5）单步执行程序（Step Into）。单步执行程序。每执行一步，所有已经打开的相关窗口的内容都更新。

（6）单步执行当前函数（Step out of）。单步执行当前函数，直到返回到它的调用程序。仅用于 C 语言程序。

（7）复位（Reset）。通过 VisualDSP++对处理器进行复位。如果是模拟器状态，IDDE

使模拟器所有状态复位；如果是硬件会话环境下，IDDE 将通过仿真器向目标系统上的处理器发送复位信号。复位后，必须重新加载程序才可以运行。

3．程序性能分析操作

Debugger 中提供两个工具来分析程序的执行情况：跟踪（Trace）和线性剖析（Linear Profiling）。这两个命令都位于 Tools 下拉菜单中。

Trace 提供对程序执行指令的跟踪，结果显示程序如何执行到某一地址上，显示程序的读、写和存储器访问。通过如下步骤来设置 Trace 功能并显示其结果。

（1）单击“Tool | Trace”，选择使能跟踪，激活跟踪操作。

（2）单击“Tool | Trace”，选择并设置跟踪深度，默认为最大跟踪深度。

（3）单击“View | Debug Windows”，选择“Trace”打开跟踪显示窗口；

（4）运行程序，在停止程序执行后（设置断点，或使用 Halt 命令），通过跟踪窗口可以查看跟踪的执行结果。

跟踪结果含有如下内容：访问类型（RD 或 WR）、内存类型（PM 或 DM）等。对于跟踪分析工具的使用需要注意：对 Blackfin 系列处理器，模拟器中不支持跟踪，仿真器中才支持跟踪。

剖析工具 Linear Profile 用来分析程序的运行时间特性。通过线性统计剖析，可以分析出每段程序的耗时量和在整个程序运行中所占用的比例，为用户分析程序的性能、优化程序提供帮助。VisualDSP++对运行的程序做统计分析，计算出每条指令占用执行程序中的百分比和运行的周期数，并以统计表的形式给出。完成一次剖析的基本步骤如下。

（1）编译和链接工程完成。

（2）单击“Tool | Linear Profiling”，选择“New Profiling”建立和激活新剖析。

（3）在新建剖析窗中空白处单击鼠标右键，选择“Properties…”。

剖析参数设置窗口中，可以对全部程序进行分析，也可对 C/C++子函数进行分析，还可以指定程序的地址段进行分析。

（4）设置结束后，运行程序。在剖析窗口中将显示线性统计剖析的结果。

剖析窗口左半部分为用户所分析的程序段相关的结果，用鼠标双击相关函数，窗口右半部分将显示对该函数中每条指令进行线性统计的结果。右半窗口中显示结果默认为每个子程序或每个程序占整个剖析运行程序段的百分比。通过修改剖析参数，也可显示每条指令执行的周期数。

值得注意的是，由于会使用到预编译器，每次编译后剖析窗口中的数值都会发生变化。另外，如果对剖析窗口中的数值不采用清除操作，剖析结果将一直累计。所以，无论程序重新编译还是重新执行，都应当先将剖析窗口中原有数值清除后再运行程序进行分析，否则分析出来的数据有可能不准确。

4．设置观察点

观察点（Watch Point）与断点功能非常相似，断点可以在程序的任意位置上设置，使程

序暂时停止执行。而观察点可以设置某种条件，当满足条件时才暂停程序的执行，如存储器读写、堆栈弹出等。通过下列步骤来设置观察点。

（1）选择“Settings | Watch points”，弹出 Watch points 窗口。

（2）在该窗口中设置程序停止条件，有 3 种判断方式：寄存器、硬件堆栈和存储器。下面以寄存器页面为例说明，主要包括如下一些设置。

Register：列出所有寄存器，为用户提供选择所需的寄存器。

Watch For Read：寄存器读操作满足条件，就暂停程序执行，如读到任意值、读入特定值、读入值为某计算的操作数或读入未定义值。

Watch For Write：寄存器写操作满足条件，就暂停程序执行，如写任意值、写指定值、写的值作为某种计算的操作数或写未定义值。

Value：读或写操作的指定值。

Format：读或写的指定值的格式，可选二进制、整数、浮点数等。

Add、Edit、Delete：对观察点列表进行添加、删除、编辑等操作。

（3）对观察点判断条件设置完毕后，单击“Add”按钮，将所指定的观察点加入到观察点列表中。利用“Add”按钮可以加入多个观察点到列表中。

（4）单击“OK”按钮，完成设置。

（5）运行程序，当满足设置的任意一个条件时，将自动停止运行程序。

5．模拟硬件环境

为方便在模拟器环境下调试程序，调试器提供了 3 种硬件环境的方式模拟：中断，模拟在程序执行过程中产生外部随机中断；数据流，模拟处理器通过外部端口进行数据传输；Load SimLoader 模拟处理器通过 EPROM 或主机等方式的加载过程。这 3 种硬件模拟均在 Settings 菜单中。

（1）中断（Interrupts）模拟

中断模拟对调试中断服务程序非常有用，其设置内容如下。

外部中断类型：包括 FLAG 中断、IRQ 中断、定时器中断等，具体中断类型与处理器型号有关，不同处理器所包含的外部中断类型有所不同。

Min cycles：中断信号产生的最小指令周期间隔。

Max cycles：中断信号产生的最大指令周期间隔。

Offset cycles：在第一次中断发生之前的指令周期数。

Interrupts：显示设置完成的模拟中断及其参数。

Add、Remove、Remove All：用于对模拟中断进行添加、删除的管理操作。

模拟中断设置后直接运行程序即可，中断模拟器将会按所设置方式产生中断。注意，重新执行、重新编译链接或重新加载程序，模拟中断都不会取消。重新启动 VisualDSP++ 可以关闭模拟中断。

（2）数据流模拟和 DMA 模拟传输（Blackfin 系列处理器）

数据流模拟可以模拟处理器的外部数据总线、链路口、串口等端口的数据传输。该功

能对于在模拟器环境下调试处理器的 DMA 传输非常有效。

选中“Stream”菜单项，弹出数据流管理窗口；单击“添加”按钮弹出新建模拟数据流传输设置窗口，在这里可以对已经存在的数据流模拟进行修改或删除等操作。新建数据流设置窗口主要包含数据流的源和目的、类型端口等，主要参数如下。

数据传送的源/目的：设置数据传输的来源和目的，只有源和目的均正确才能进行数据流传输。

处理器：给出针对数据流模拟的器件，默认为会话选择的处理器型号。

数据流使用的设备：也就是数据流传输使用的端口或者存储器。

地址：数据流使用的 I/O 地址。

文件/浏览：用于采用文件方式实现数据流模拟中打开数据文件。

数据传输使用的格式：数据流可采用整数类型和浮点格式进行数据传输。

循环方式：数据文件读取结束后是否采用循环方式再从头读取数据。

数据流设置完成后，单击“OK”按钮，退出窗口；

运行程序，在程序运行过程中，处理器通过指令或 DMA 均可以实现对数据的获取或写出。

处理器读取数据流一般是从外部文件将数据读入，处理器写出的数据也将写出到文件中，用户通过相关的文件可以查看处理器传输的数据。

用户设置完该窗口中的参数后，单击“OK”按钮即可。数据的传输由用户的程序来控制开启，并且只能采用 DMA 方式进行数据传输，处理器通过指令访问将无效。

值得注意的是，由于对数据的传输需要通过用户的程序控制开启，因此在该窗口中的设置应当与用户程序中所设置的 DMA 通道及其方向一致，否则将不能模拟 DMA 的正常传输。

（3）Load Sim Loader 模拟

Load Sim Loader 用来模拟 EPROM 或主机给处理器加载.1dr 文件的过程，为用户设计实现处理器加载提供帮助。通过下列步骤可建立处理器 EPROM 加载。

（1）选择“Settings”中的“Load Sim Loader”，从中选择从主机加载启动或从 PROM 加载启动。

（2）无论模拟主机加载启动还是 PROM 加载启动，选择以后，VisualDSP++都将提示选择所需要加载的文件（后缀是.ldr）。

（3）选择加载文件后，VisualDSP++提示用户单击调试菜单中的复位按钮，然后弹出提示信息。

（4）在用户单击确认并复位完成后，VisualDSP++将进入模拟加载过程。

（5）进入模拟加载过程后，用户直接运行程序，VisualDSP++将自动完成加载过程。用户通过单步执行程序，可以观察到处理器的模拟加载过程。

（6）如果要从模拟加载环境下退出，单击菜单“Settings”中的“Load Sim Loader”，并将其设置成无加载方式。

6．寄存器窗口操作

寄存器操作是 DSP 调试过程中常用的。寄存器的显示通过在“Register”菜单中进行选择。寄存器下拉菜单包含了内核寄存器、系统寄存器、I/O 端口寄存器或附属设备寄存器等。通过鼠标右键可更改寄存器数据显示方式，寄存器数据格式包括十六进制、八进制、二进制、有符号或无符号整数、32/40 位浮点、有符号或无符号小数等。

在寄存器窗口中可以查看寄存器内容、显示的数据格式和修改寄存器内容。修改寄存器值只需双击需修改寄存器后进行修改，再输入新值并按回车键即可。

7．存储器窗口操作

存储器窗口不但像寄存器窗口那样，可以提供数据格式和编辑操作，还提供跳转、查找、填充、导出等功能。下面分别介绍存储器操作。

存储器查看：选中菜单“Memory”选择存储器查看方式，弹出存储器窗口。

改变存储器数据格式：类似对寄存器窗口的操作。

跳到某一地址上查看：可以直接在存储器窗口的跳转地址栏中输入所需跳转的地址即可，该地址栏支持十六进制地址输入和符号选择；也可以在激活的存储器窗口上单击鼠标右键，在菜单中选择“Go To”命令，在窗口中输入十六进制的地址或通过 Browse 从标号列表中选择一个符号，单击“OK”按钮即可。

填充或者导出存储器数据：填充是把数据填充到存储器中，导出是把存储器内容写到数据文件（.dat）中。

填充：在激活的存储器窗口中单击鼠标右键，在菜单中选择“Fill”，弹出存储器数据填充窗口。配置该窗口中的参数，完成后单击“OK”按钮即可。填充存储器的数据来源有两个：固定值和用户数据文件。

导出：在激活的存储器窗口中单击鼠标右键，在菜单中选择“dump”，弹出存储器数据导出设置窗口。配置该窗口中的参数，完成后单击“OK”按钮即可。

新建跟踪：可在某一存储器窗口中输入表达式来进行跟踪，通过下列步骤来跟踪表达式：在激活的存储器窗口中单击鼠标右键，选择菜单项“New-Tracking”，弹出“输入新跟踪表达式”窗口；输入表达式，可以是 C 或寄存器表达式，寄存器表达式必须用$Xn 的形式；单击“OK”按钮。

将存储器内容画图：存储器中的数据也可用图形的形式给出，基本步骤如下。

（1）新建一个画图窗口：选择菜单项“View | Debug | Plot”，单击“New”按钮。

（2）在弹出的新建画图配置窗口中设置如下选项：

数据集合；添加/删除/创建按钮。

绘图类型：常用的有线段图、X—Y 图、星座图、眼图、谱图等。

存储器类型：绘图数据存放的存储器类型。

存储器地址：绘图数据存放的存储器起始地址，可以用直接地址或变量名。

偏移地址：绘图数据存放的地址与存储器地址中定义的地址之间的偏移量。

此外还有标题、图形名称、数据长度、地址增量、数据类型和绘图的高级设置等。

（3）绘图配置完成后，单击“Add”按钮，将设置完成的绘图添加到数据集合中。

（4）单击“OK”按钮，VisualDSP++将显示所绘制的图形。

在图形窗口中，可以直接使用鼠标选择对区域图形进行放大，通过鼠标右键菜单中的“Reset Zoom”可以将图形恢复到满窗口模式。在图形窗口的鼠标右键菜单中有光标，用户通过移动光标可以在图形窗口的左下角查看到光标位置所对应的数据序号和数值的大小。图形窗口允许用户通过导出的方式将图形以图片或者数据的方式进行保存。单击鼠标右键菜单中的“导出”，将弹出导出图形设置窗口。

关于 VisualDSP++使用的更多知识，请参考其用户手册或其他参考书籍[31~32]。

第4章　基于Blackfin处理器的最小视频系统

在利用 ADI 公司 Blackfin DSP 处理器开发更为复杂的数字视频处理系统之前，首先需要掌握基于 Blackfin DSP 的数字视频处理系统的构成和基本处理过程，包括数字视频的输入/输出，对视频数据进行简单处理所需要了解的相关知识，以及开发框架和开发流程等。通过本章的学习，读者将能够搭建一个数字视频处理的最小系统，其中包括视频输入、输出和对视频内容的简单处理。本章最后介绍了简单视频处理系统的实现。

Blackfin 处理器架构非常适于多媒体系统设计，其功能丰富的外设为音视频设备提供众多连接选项。此外还提供了许多辅助工具，能够帮助用户快速完成系统设计。这些工具包括 VisualDSP++工具套件、EZ-KIT 评估板和 EZ-Extender 扩展卡等。VisualDSP++工具套件提供了许多外设驱动程序，适用于 PPI 接口、TWI 接口、定时器及连接视频系统所需要的所有设备。另外还有一套完善的视频设备驱动程序，不仅有面向模数和数模转换器的驱动程序，还有面向 CMOS 摄像头和 LCD 显示器的驱动程序等。

本章结合 ADI 公司的 BF533 和 BF561 DSP 处理器及其相应的 Blackfin EZ-KIT 开发套装来进行介绍。Blackfin EZ-KIT 开发板集成了视频编码器和视频解码器，分别支持视频数据采集和回放功能，极大地方便了基于 Blackfin 视频应用的快速开发。

文中首先对数字视频处理系统的构成和设备情况进行简要介绍；然后介绍 BF533/BF561 DSP 处理器及其 EZ-KIT 开发板，了解它们提供的各种功能模块和接口；接着介绍了视频输入和视频输出的硬件连接方式，以及视频输入、输出的基本实现过程和代码；最后在此基础上实现一个简单的视频处理算法。

4.1　数字视频处理系统构成

数字视频处理系统主要由 3 部分组成：视频源、处理器及视频输出设备。图 4.1 描述了一个端到端的数字视频系统。其中，视频源负责提供原始的视频数据；处理器（假设具有编解码功能）模块完成对输入视频数据的压缩编码，或者将压缩过的码流解压，并直接传输到某种视频输出设备，如 TFT-LCD 平板；显示设备则负责将视频结果呈现给最终用户。

处理器的数字视频处理功能是通过软件来实现的，其处理性能不仅受限于处理器本身的特性，如 CPU 的工作频率、片上和片外内存的大小、DMA 控制器等，而且还受限于视频源与处理器、处理器与视频输出设备之间的数据传输效率。Blackfin 处理器上的 PPI 接口是负责传输视频信号的端口。PPI 即并行外设接口（Parallel Peripheral Interface），它不仅是一个视频接口，而且还可支持高速并行转换器。它是一个高速并行端口，同时还具备一些面向视频信号的出色特性。这就为视频的高速传输提供了硬件基础。

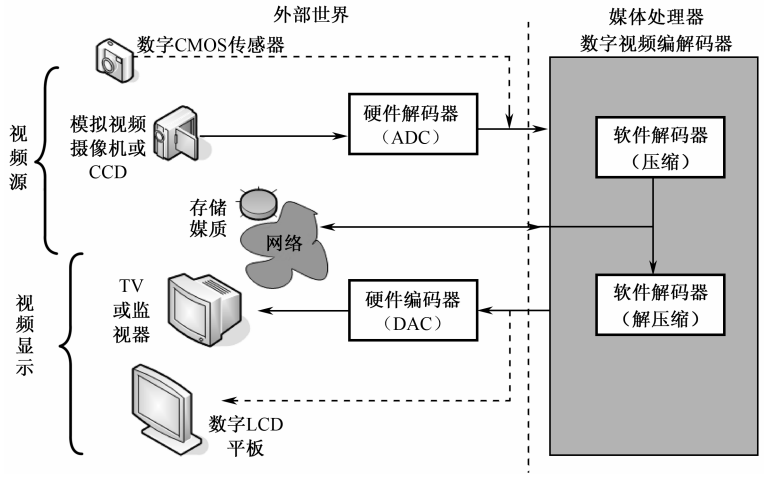


图 4.1　数字视频处理系统构成

视频源又分为模拟视频源和数字视频源两类。对于模拟视频源，数字处理器内核无法直接对其信号进行处理。视频信号必须首先通过视频解码器数字化，将模拟视频信号（如 NTSC、PAL、CVBS、S-Video）转换为数字信号形式（通常是 ITU-R BT.601/656 YCbCr 或者 RGB）。这是一个复杂的、多级的处理过程，包括从输入信号中提取时间信息、亮度与色度的分离，色度信息分离为 Cr 和 Cb 分量，输出数据的采样及为其分配适当的格式等。通过串行接口，如 SPI 或 I^2C，可以对解码器的操作参数进行配置。如图 4.2 所示是典型视频解码器的结构框图。

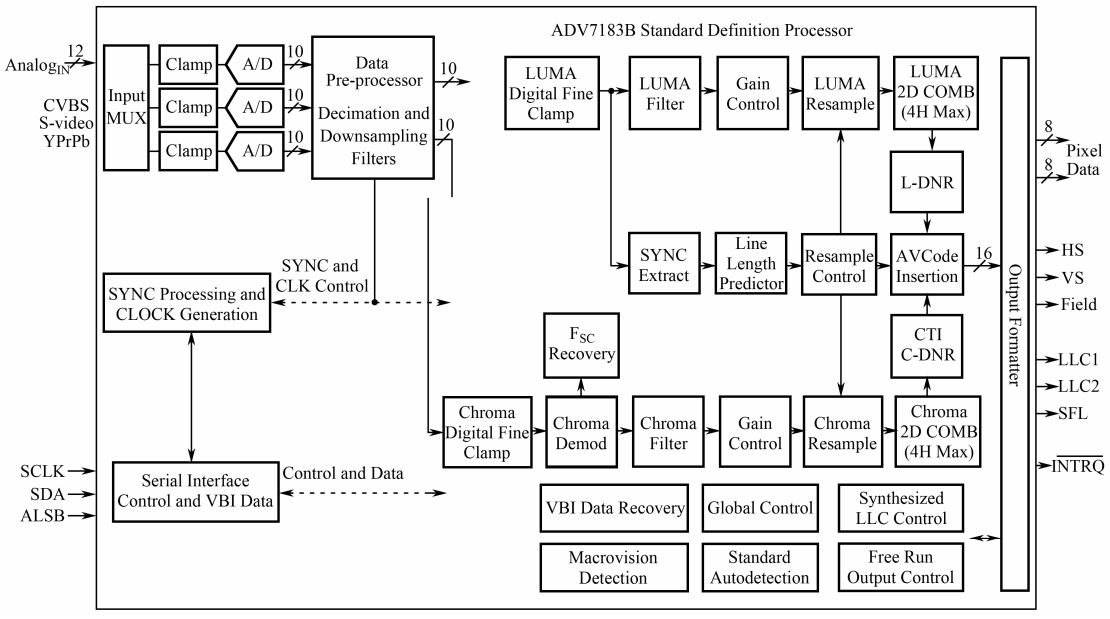


图 4.2　典型视频解码器的结构框图

合乎标准的数字视频源信号可以直接通过 PPI 接口进入处理器。当今的数字视频信号源基本上都是基于电荷耦合设备（CCD）或 CMOS 技术，它们可以将光转换为电信号。CMOS 传感器一般会输出并行的数字信号流，该信号流通常包括 YCbCr 或 RGB 格式的像素分量，以及水平/垂直同步和像素时钟。有时，它们还允许采用一路外部的时钟和同步信号，以控制图像帧从传感器向外部传输。CCD 往往连接到“模拟前端”（AFE）芯片上，如 AD9948，该类芯片处理模拟输出信号，对其进行数字化，并产生恰当的时序信号来扫描 CCD 阵列。AFE 的同步信号由处理器提供，AFE 需要利用该路控制信号来管理 CCD 阵列。

视频显示设备也分为模拟和数字两类。对于模拟视频显示，需要借助视频编码器将数字视频流转换为一路模拟视频信号，其输入一般为 ITU-R.656 或 BT.601 格式的 YCbCr 或 RGB 视频流，根据不同的输出标准（如 NTSC、PAL、SECAM）对信号进行转换。主控处理器可以通过 2 线或 3 线串行接口（SPI 或者 I^2C）来对编码器进行控制，如对像素时序、输入/输出格式及亮度/色度滤波等设置进行编程。如图 4.3 所示是典型视频编码器的结构框图。

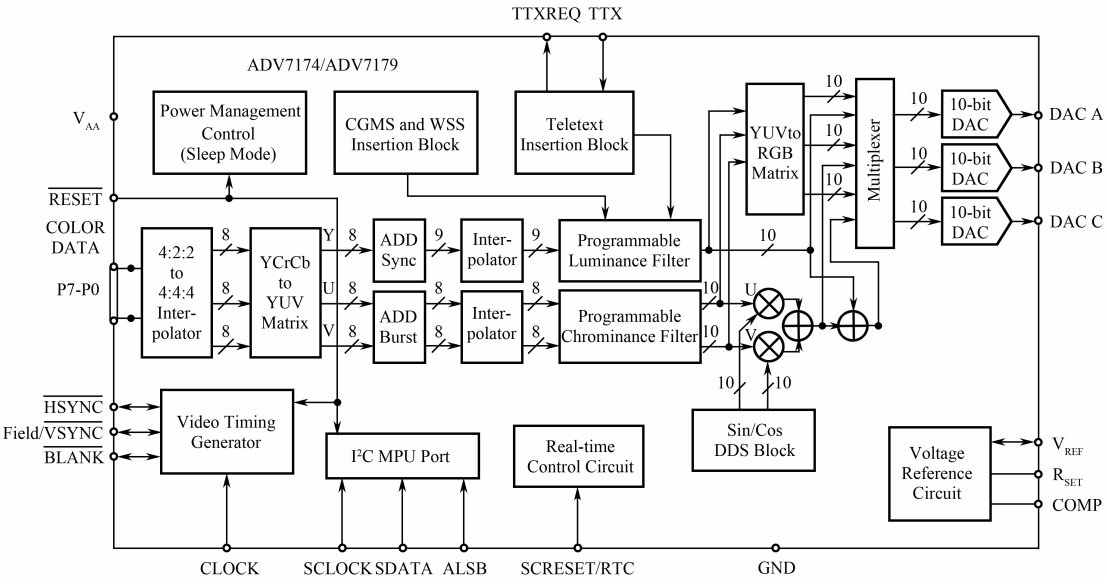


图 4.3　典型视频编码器的结构框图

视频编码器比较常见的模拟输出格式包括复合视频基带信号（CVBS）、S 端子视频、分量视频及模拟 RGB 信号。复合视频将亮度、色度、同步和色彩脉冲信息整合到一根电缆内。S 端子视频分别传送亮度和色度内容，将亮度信息与色差信号分离开可大幅改善图像质量，因而在家庭影院系统中比较流行。分量视频中，每个亮度与色度通道都是单独提取、输出的，每路都带有自己的时序，保证了模拟传输后图像的高品质，常见于高端家用影院系统组件，如 DVD 播放器和 A/V 接收机。模拟 RGB 具有分离的红、绿、蓝信号通道，可以提供类似于分量视频的图像质量，但一般用于计算机图形图像领域；RGB 连接器往往是 BNC 插座的改型。

如图 4.4 所示为上述各种输出格式的模拟视频连接器。

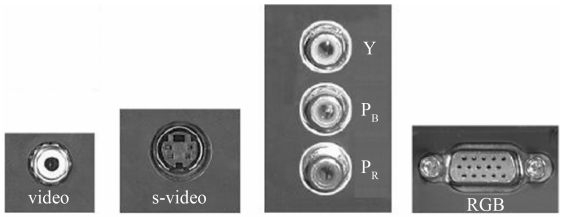


图 4.4　常见的模拟视频连接器

传统的计算机监视器采用阴极射线管（CRT）技术，用 3 根独立的插针调制三相电子枪波束，电子束激励屏幕上的磷点，则该点将发出红、绿、蓝或三色的组合，具体的发光则取决于是哪路电子束击中了该点。CRT 的主要优势是廉价，可以产生的颜色也要比同等尺寸的 LCD 平板丰富。此外，CRT 的可视角度非常宽广。CRT 的不足在于体积大、电磁辐射强，由于刷新时的闪烁效应，也会导致眼睛疲劳。

现在常见的是液晶显示（Liquid Crystal Display，LCD）平板。LCD 技术主要有两类：无源阵列和有源阵列。无源阵列往往采用由一块印刷出行电极引线结构的玻璃衬底与另一块印刷出列电极引线结构的玻璃衬底共同组成的“液晶三明治”结构。这些行列交叉点构成了像素点。为了激活特定的像素，时序电路将像素点处的列上电，而将行接地。所形成的电压差则使得该像素点上的液晶产生翻转，于是该点变得不透明，阻止光线穿过。无源阵列技术简单，但存在一些缺点，如刷新时间较慢、行—列交点处电压容易泄漏到相邻像素点、可视角度相对较小。

有源阵列 LCD 从基本结构来看，每个像素点都由一个电容和晶体管开关构成，即称为“薄膜晶体管（TFT）显示”的结构。为了对特定的像素进行定位，需要使能其所在行，然后向其所在列施加一个电压，这样可以带来隔离感兴趣的像素点的效果，而周围的其他像素都不会被影响。由于控制特定像素的电流被降低，该像素点的开关速度也更高，这就使得 TFT 技术具备了高于无源显示的刷新速率。此外，对施加到像素点上电压高低的调制也可以实现对多种亮度级的显示。如今，对于 8 位的亮度信息，其显示的亮度级通常有 256 个。

4.2　Blackfin 处理器与评估板简介[38,39]

在视频处理方面比较典型的 Blackfin 处理器有 BF533 和 BF561。其中后者是双核处理器，因而能够同时实现视频的输入和输出操作，同时具有更高的处理性能和更多的内存，特别适合于高端的视频处理应用中。BF533 则由于其成本优势适用于中低端市场。本章主要以这二者为例介绍最小视频处理系统的实现。针对不同的 Blackfin 处理器，ADI 都提供了对应的评估套件，为用户快速完成对处理器性能的评测并搭建原型系统提供了极大的方便。下面首先对 BF533/561 处理器和评估板进行简单介绍，了解搭建视频处理系统的硬件基础。

4.2.1　ADSP–BF533：高性能的通用 Blackfin 处理器

ADSP-BF531/ADSP-BF532/ADSP-BF533 处理器是 Blackfin 产品家族的成员，采用了 ADI 公司和 Intel 公司提出的微信号结构（MSA）。Blackfin 处理器融合了一个双 MAC 的先进信号处理引擎，在单个指令集合结构中实现了干净的、正交的类 RISC 微处理器指令集合，同时具有单指令多数据（SIMD）多媒体处理能力。以上几种处理器完全代码和管教兼容，差别仅在于其性能和片上内存。通过集成一组丰富的业界领先的系统外设、内存，Blackfin 处理器是需要类 RISC 编程能力的新一代应用的理想开发平台，在一个集成包上支持多媒体和先进的信号处理。

ADSP-BF533 为当今高要求的汇聚性信号处理应用提供了高性能、低功耗处理器选择。高性能的 16 位/32 位 Blackfin 嵌入式处理器内核，灵活的高速缓存结构，增强的 DMA 子系统，以及动态功耗管理（DPM）功能为系统设计者提供了灵活的平台，能解决很大范围的应用，包括消费类、通信、汽车和工业设备等产品。

ADSP-BF533 功能框图如图 4.5 所示，其特性如下：

（1）高性能的 16 位/32 位嵌入式处理器内核。

（2）10 阶段 RISC MCU/DSP 流水线，支持混合 16 位/32 位 ISA 以优化代码密度。

（3）完全的 SIMD 结构，包括加速视频和图像处理的指令。

（4）内存管理单元（MMU）支持完整的内存保护。

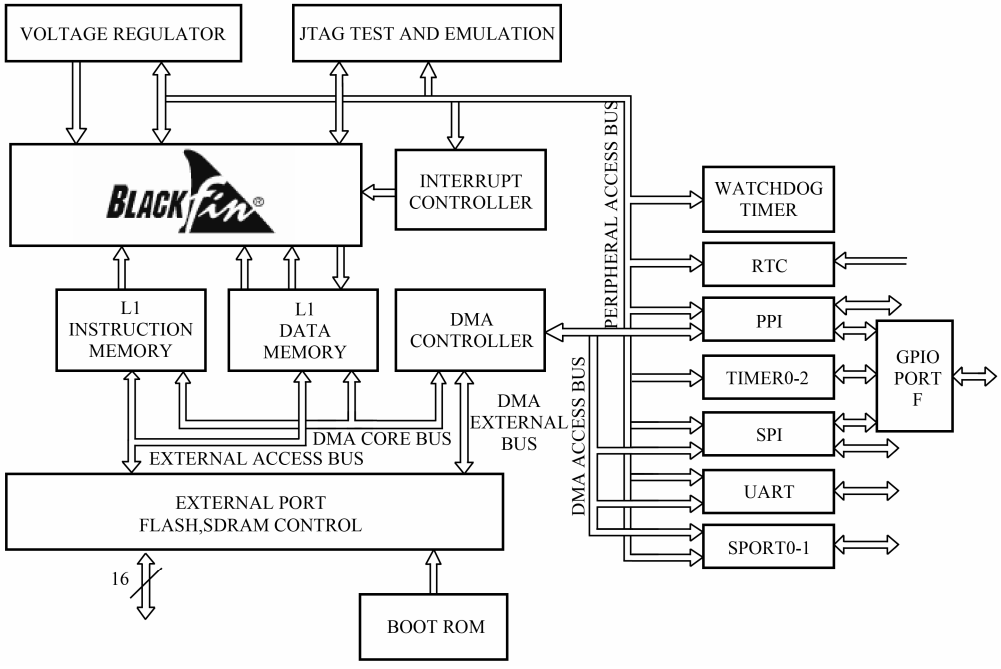


图 4.5　ADSP-BF533 功能框图

集成高级特性如下：

（1）148KB 片上 SRAM。

（2）PPI 接口，支持 ITU-R 656 视频数据格式。

（3）两个双通道、全双工的同步串行端口，支持 8 个立体声 I^2S 通道。

（4）12 个 DMA 通道，支持一维和二维数据传输。

（5）内存控制器，提供到多个外部 SDRAM、SRAM、Flash 或 ROM 的无缝连接。

（6）两个内存到内存的 DMA，8 个外设 DMA。

（7）3 个 32 位定时/计数器，支持 PWM。

（8）实时时钟和看家狗定时器，32 位内核定时器。

（9）16 个通用 I/O 引脚（GPIO）。

（10）片上 PLL 支持倍频。

4.2.2 ADSP–BF561：用于消费者多媒体的 Blackfin 对称多核处理器

ADSP-BF561 扩展了 Blackfin 处理器家族的性能边界，具有两个高性能的 Blackfin 处理器内核、灵活的高速缓冲结构、增强的 DMA 子系统及动态功率控制功能，能够支持复杂的控制和信号处理任务，保证极高的数据吞吐量。ADSP-BF561 非常适合应用于众多工业、仪器仪表、医疗和消费类电子等，对终端产品所需的数据带宽支持可扩展性。

ADSP-BF561 功能框图如图 4.6 所示，主要具有如下相关特性。

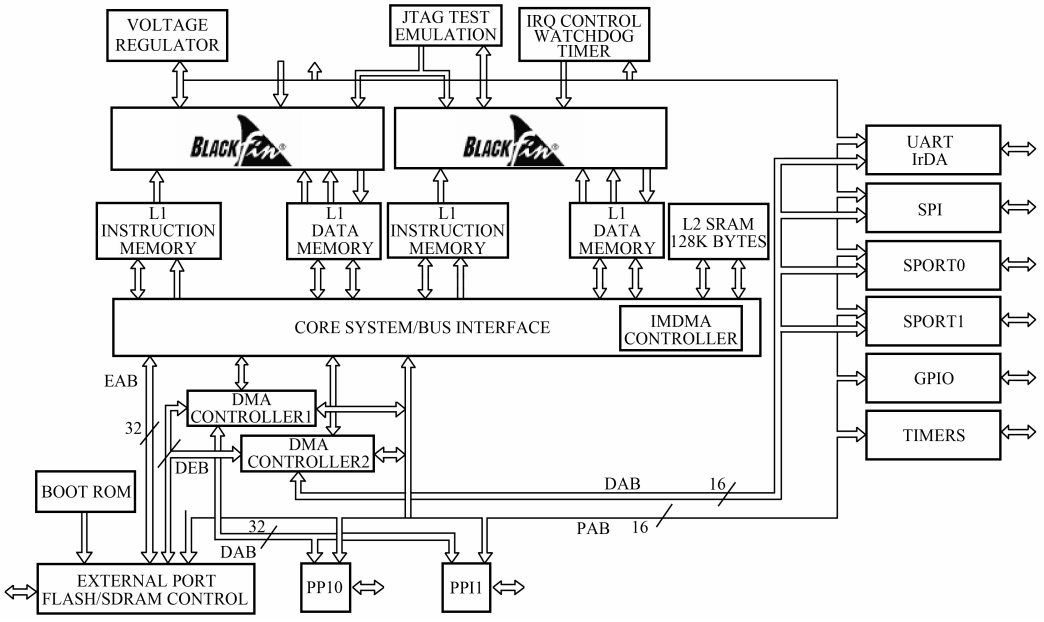


图 4.6　ADSP-BF561 功能框图

（1）最高时钟速率为 600MHz（2400 MMACS）。

（2）328KB 的片上内存，其中每个内核有 32KB 的 L1 指令内存 SRAM/Cache，每内核

有 64KB 的 L1 数据内存 SRAM/Cache，每内核有 4KB 的 L1 中间结果暂存器内存；同时还有共享的 128KB 低延迟的 L2 内存。

（3）32 位内存控制器，提供到多个 SDRAM、SRAM、Flash 或 ROM 的无缝链接。

（4）两个 PPI 接口单元，支持 ITU-R 656 视频数据格式。

（5）两个双通道、全双工同步串行接口，支持 8 个立体声 I^2S 通道。

（6）双 16 通道 DMA 控制器，支持一维和二维传输。

（7）SPI 兼容端口。

（8）12 个定时器/计数器，支持 PWM、脉冲宽度和时间计数模式。

（9）48 个可编程标志（GPIO）。

（10）双看家狗定时器。

（11）锁相环 PLL 支持 1×～63×的倍频。

4.2.3　EZ−KIT Lite for ADSP−BF533

ADSP-BF533 EZ-KIT Lite（如图 4.7 所示）为开发者评测 Blackfin 处理器、快速实现原型提供了低成本方案，相关的应用包括音频和视频处理等。EZ-KIT Lite 包括一个 ADSP-BF533 桌面评估板，以及一个基本的调试软件来降低结构评价的难度。其主要特性如下：

（1）ADSP-BF533 Blackfin 处理器。

（2）64MB（32M×16-bit）SDRAM。

（3）2MB（512K×16-bit×2）Flash 内存。

（4）ADV7183 视频解码器，带 3 个输入 RCA 插口。

（5）AD1836 96kHz 音频编解码器，带 4 个输入和 6 个输出 RCA 插口。

（6）ADV7171 视频编码器，带 3 个输出 RCA 接口。

（7）ADM3202 RS-232 线驱动器/接收器。

它同时还支持各种 Blackfin EZ-Extender 扩展子板，如音视频扩展板 Blackfin A-V EZ-Extender（如图 4.8 所示），该扩展板上有 AD1836，一个 96kHz 音频编解码器；5 个 3.5mm 的音频插口；视频编码器 ADV7179；视频解码器 ADV7183B；以及到平板显示接口的链接。不同的扩展板能够对系统提供不同方面的功能扩展。

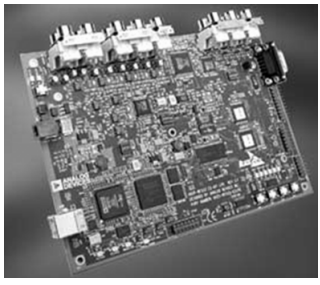


图 4.7　ADSP-BF533 EZ-KIT Lite

图 4.8　音视频扩展板

4.2.4 EZ−KIT Lite for ADSP−BF561

ADSP-BF561 EZ-KIT Lite（如图 4.9 所示）通过使用一个基于 USB 的以 PC 为主机的工具集合，为开发者提供一个对 ADSP-BF561 处理器进行初始评测的低成本方法，主要面向音视频应用。对模拟音频应用的评测是通过使用 AD1836 多通道 96kHz 音频编解码获得的。通过使用 ADV7183A 高级 10 位视频解码器和 ADV7179 芯片级 NTSC/PAL 音频编码器，用户能够对音频应用进行评测，如同时进行视频输入和输出处理——这是由 ADSP-BF561 处理器的双核结构提供的支持。利用该 EZ-KIT Lite，用户能够学习到 ADSP-BF561 硬件和软件开发的更多知识，并快速完成应用程序的原型化过程。

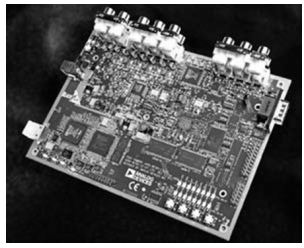


图 4.9　ADSP-BF561 EZ-KIT Lite

EZ-KIT Lite 包括一个 ADSP-BF561 处理器桌面评价板，以及一个 VisualDSP++开发和调试环境评价套件，包括 C/C++编译器、汇编器和链接器。它还包含处理器应用的例子程序、电源线和 USB 电缆。

该 EZ-KIT Lite 的特性如下：

（1）ADSP-BF561 Blackfin 处理器。

（2）64MB（16M×16 位×2）SDRAM。

（3）8MB（4M×16 位）Flash 内存。

（4）AD1836 多通道 96kHz 音频编解码器。

（5）3 个 RCA 插口，用于复合视频（CVBS）、差分视频（YUV）或 S 视频（Y/C）输入。

（6）立体声音频输入/输出 RCA 插口。

（7）ADV7183A 高级 10 位视频解码器。

（8）ADV7179 芯片级 NTSC/PAL 视频编码器。

（9）3 个 RCA 插口，用于组合视频（CVBS）、分量视频（RGB）、差分视频（YUV）或 S 端子视频（Y/C）输出。

同时还有针对它的一系列 EZ 扩展板，如 Blackfin 音视频 EZ 扩展子板、Blackfin FPGA EZ 扩展板、Blackfin 音频 EZ 扩展板、Blackfin USB-LAN EZ 扩展板等。

4.3　Blackfin 处理器与视频外设之间的连接

Blackfin 处理器提供享有全面技术支持的硬件和软件模块，能够帮助用户快速开发在

Blackfin 处理器上运行的视频应用。本节将介绍如何将视频设备连接至 Blackfin 处理器，以及连接过程中所必须遵循的基本原则。图 4.1 给出了视频源、视频编解码器、DSP 处理器之间的连接关系和基本的视频流过程。接下来介绍 Blackfin 处理器的视频接口 PPI（并行外设接口），以及如何通过 PPI 接口将处理器与视频外设相连接。在 Blackfin 的评估板和/或扩展板上，已经支持下面要介绍的各种视频信号的连接方式。在设计自己的产品电路时，参考评估板和相关扩展板的设计电路是非常有帮助的。

4.3.1　Blackfin 处理器上的视频接口——PPI

Blackfin 处理器 PPI 接口是负责处理视频信号的视频端口。PPI 的意思是并行外设接口，它不仅是一个视频接口，具备一些面向视频信号的出色特性，而且还可支持高速并行转换器。

PPI 是一个相当简单的接口，包含 16 个数据线路，即图 4.10 中的 PPI15～PPI0。最多可以传输 3 个帧同步信号：PPI_FS1、PPI_FS2 和 PPI_FS3，然后是时钟信号 PPI_CLK。时钟信号 PPI_CLK 始终是从外部设备通过 PPI 接口输入 Blackfin 处理器。帧同步信号和视频数据则取决于当前的运行模式，既可以是输入，又可以是输出。需要指出的是，在 ITU 656 运行模式下，不需要使用这些帧同步线路，因为数据流包含了所有的同步信号。参考后面几个例子可以更好地了解这几个接口的使用方法。

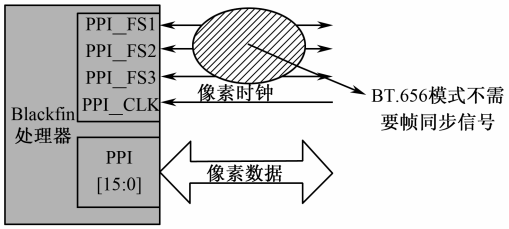


图 4.10　PPI 接口

PPI 是一个双向的半双工接口，在特定时刻既可执行视频输入，又可执行视频输出。PPI 接口支持 ITU 656 建议中规定的位并行模式。时钟信号和帧同步信号具备一定的信号极性可编程性。

PPI 接口拥有大量节省带宽的特性，如用户可以选择仅接收 ITU 656 输入视频帧中的一部分信号。例如，用户可以选择仅接收有效视频信号而忽略消隐区域，忽略消隐区域的意思是 PPI 接口不将消隐信号传输至 DMA 控制器。这样一来既节省了 DMA 带宽，又减轻了 Blackfin 处理器中的数据传输任务。如果用户只需要查看消隐信号，也可以利用这个特性。例如，在某些类型的隔行电视应用或封闭式字幕应用中，用户只需要查看视频帧的消隐信号区域。当然，也可以接收完整的视频帧，包括消隐信号和有效视频信号。

用户还可以选择忽略一个隔行视频流中的场 2 信号。在一些对数据或视频质量要求不高，但需要进行高质量视频压缩的应用中，就可以利用这个特性，仅处理每个视频帧中的

场 1 信号。用户也可以选择仅处理奇或偶数据元，仅读取 4:2:2 输入视频流中的亮度或色度信号。整个 PPI 外设与 Blackfin 处理器的二维 DMA 控制器携手合作，允许用户指定任意内存区域，并确保仅将其中的数据发送至 PPI 接口。用户甚至可以指定将一幅图像的任意矩形区域发送至 PPI 接口。

4.3.2 将 Blackfin 处理器连接至视频源[50]

图 4.11 显示了将 Blackfin 处理器连接至数字视频源的连接方式。其中 CMOS 摄像头的 I^2C 总线与 Blackfin 处理器的双线接口 TWI 之间形成了控制通道。该控制通道主要用于配置视频源。在图 4.11 的例子中，视频源提供的 8 位数据流将输入 Blackfin 处理器上的 8 位 PPI 接口（PPI7～PPI0）。用户完全可以实现 10 位、12 位甚至 16 位的连接，这取决于所用摄像头的分辨率。图 4.11 中显示的是一个 8 位连接的例子。

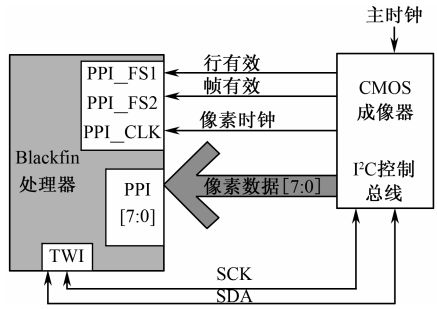


图 4.11　将 Blackfin 处理器连接至数字视频源

在如图 4.11 所示的数字视频源情况下，一般是由摄像头提供像素时钟信号和执行某种类型的成帧处理。水平同步信号将标定有效扫描行区域，而垂直同步信号则是一个“帧有效”类信号。对于支持 BT.656 标准的摄像设备，则不需要这些同步信号。

Blackfin EZ-Extender 扩展卡和 EZ-KIT 评估板可以支持众多先进制造商提供的范围广泛的 CMOS 摄像头，包括 Micron、Omnivision 和柯达公司等。如图 4.12 所示是 Micron 公司提供的 CMOS 摄像“头板”。它具备一个通用接口，可以全面支持各种型号的摄像头，该设计提高了 CMOS 摄像头连接的通用性。

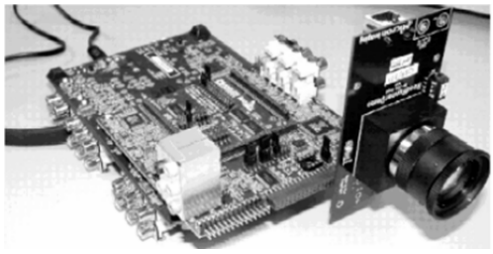


图 4.12　Micron 公司的 CMOS 摄像头板

为了将模拟可携式摄像机提供的视频信号输入 Blackfin 处理器，首先要利用诸如

ADV7183B 等视频解码器，将模拟视频信号转换为数字信号。然后与前面类似，ADV7183B 再通过 I²C 总线与 Blackfin 处理器的双线接口 TWI 之间建立起控制通道，以便对视频源进行配置。同时 ADV7183B 解码器还是通过 PPI 接口向 Blackfin 处理器输入 8 位数字视频流。在如图 4.13 所示例子中，视频解码器 ADV7183B 向 Blackfin 处理器提供同步信号，并向其 PPI 接口输入线路锁定时钟信号。

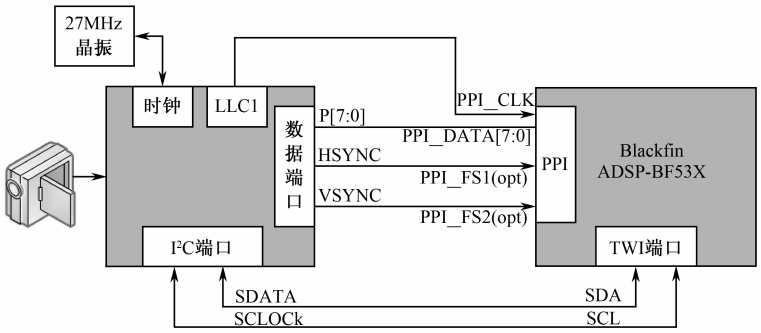


图 4.13　将 Blackfin 处理器连接至模拟视频源

4.3.3　连接至显示设备[49]

如果要将保存在内存中的视频帧传输至一台模拟显示设备进行显示，首先要借助诸如 ADV7174 或 7179 等视频编码器进行处理，将数字信号转换为符合标准的模拟信号。在连接过程中，首先利用 I²C 接口完成对视频源的配置；然后通过 PPI 数据总线，向显示设备输出视频信号；如有需要，还可通过 PPI 帧同步线路，输出帧同步信号。图 4-14 中所示例子所用的编码器无需帧同步信号，而且许多符合 BT.656 建议标准的视频编码器都不需要帧同步信号。此外，本例中是由外部定时器向编码器和 Blackfin 处理器提供时钟信号（27MHz 振荡器）。而在视频源的连接中，则是由视频解码器直接向 PPI 接口输入线路锁定时钟信号（如图 4.14 所示）。

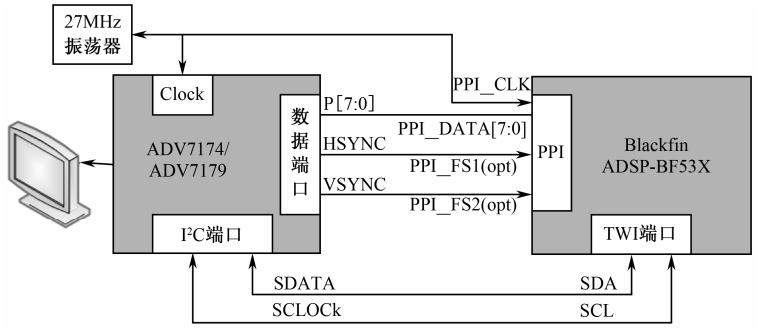


图 4.14　将 Blackfin 处理器连接至模拟显示设备

下面介绍如何将 Blackfin 处理器连接至 TFT-LCD 平板显示器。如图 4.15 所示，由 Blackfin 处理器通过 PPI 帧同步信号向 LCD 平板显示器提供水平同步信号、垂直同步信号，

由外部定时器向 PPI 时钟接口和 LCD 提供时钟信号。最下面的接口是数据总线。在连接至 TFT-LCD 平板显示器时，一般都会利用整个 16 位数据总线，因为大多数 TFT-LCD 平板显示器都可支持 18 位甚至更高数据率。在本质上是通过 16 位 PPI 接口向 18 位平板显示器输出视频信号。

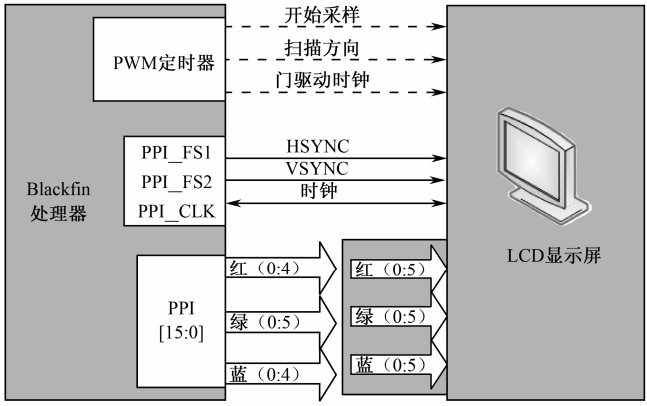


图 4.15　将 Blackfin 处理器连接至数字 TFT-LCD 面板

图 4.15 中最上面的 PWM 定时器是一个可选件。许多时候，出于材料成本或占用空间的考虑，制造商不会为 TFT-LCD 平板显示器配备定时控制器。也就是说，用户在购买 TFT 显示器时还要单独购买定时 ASIC，从而增加了系统成本。如果 TFT 显示器不具备定时控制器，那么多数情况下，Blackfin 处理器将通过外接定时器提供定时信号。按从上到下的顺序分别是：开始采样信号、标定扫描方向的信号及门驱动器时钟信号。

4.3.4　连接视频源和显示设备的原则和技巧

首先，应当尽可能采用 BT.656 标准，因为它可以大幅减少在视频应用中屡见不鲜的定时不一致问题。如果可以选择，一定要毫不迟疑地采用 BT.656 标准。此外，还需要注意所用模数或数模转换器的默认设置。有时转换器的默认运行模式未必符合用户的要求，虽然大多数时候用户都可以直接使用这些转换器，无需通过 I^2C 或 SPI 接口进行设置，但是，也有很多时候，用户需要通过 I^2C 接口的反向通道验证转换器的配置是否满足应用的需要。

另外，不论执行任何操作，必须确保尽可能不要影响时钟脉冲源。这取决于应用性能，多数应用的各种时钟频率相当高，从几 MHz 到数十 MHz。一个完全独立的时钟信号有助于提高应用系统的性能和稳定性。

最后，在将 Blackfin 处理器连接至一台 666 LCD 平板显示器时，由于这种显示器可以支持 18 位视频信号，其中红色、绿色和蓝色分量各占 6 位。Blackfin 处理器会将其视作 RGB565 显示器。但是，切勿将红色和蓝色通道中的最低有效位弃之不顾。这种通用做法会影响视频信号的动态范围。相反的是，用户可以在平板显示器上分别将红色和蓝色分量的最低有效位与最高有效位结合起来，这样就可以确保 3 个色彩分量都能提供从最低到最高

的完善的动态范围。而绿色通道则连接至 6 位接口，用于传输 6 位绿色分量信号，因为绿色是这 3 种色彩分量中对视觉影响最大的颜色。

此外，EZ-KIT 评估板、扩展卡及代码范例有着不可估量的重要价值，尤其是对于视频应用，应用框架可谓举足轻重，数据传输和存储存取效率对整个系统性能有着至关重要的影响。在硬件方面，EZ-KIT 评估板和 EZ-Extender 扩展卡，以及板载 CMOS 摄像头和 LCD 接口对用户进行系统设计起着很好的指导作用，同时可以极大地降低用户进行算法测试的代价。

4.4　数字视频信号标准简介[19~21]

ITU 关于数字电视的 BT.601 建议中，从电视广播的角度阐明了如何对数字电视信号进行编码。该标准支持 RGB 和 YCbCr 两种颜色空间。RGB 直观明了，但不适于进行视频压缩。RGB888 格式就表示红色、绿色和蓝色分量各占 8 位，RGB666 格式则表示 3 个色彩分量各占 6 位，而 RGB565 格式则表示红色和蓝色分量各占 5 位，而绿色分量占 6 位。

YCbCr 色彩空间是目前最常用的，也是 BT.601 建议中的首选。Y 表示亮度分量，Cr 或 Cb 表示色度分量。这些值是根据 RGB 值计算出来的，相互独立，因而比 RGB 信号更适于进行压缩。这也正是众多制造商纷纷选择采用这种色彩空间，BT.601 建议也是推选它的主要原因之一。

BT.601 建议中明确规定，YCbCr 4:2:2 适用于电视广播应用。它意味着要对色度值进行二次取样，每个像素取一个亮度值和一个色度值（Cr 或 Cb）。BT.601 建议支持对色彩空间分量值进行 8 位或 10 位量化。BT.601 建议的最终结果是将 NTSC 和 PAL 制式标准化为每行包含相同数量的有效像素，即 720 个有效像素。由于 PAL 信号的刷新率为 50 场/s，而 NTSC 信号的刷新率则为 60 场/s，所以通过在 PAL 信号中添加扫描行，实现了帧刷新率的标准化。

与数字视频应用相关的基本定时信号是 Hsync，即水平同步信号。该信号标定了视频帧中每个扫描行（从左至右）的有效视频信号起点。Vsync 是垂直同步信号，从上到下标定了新视频帧的起点。场是隔行扫描视频独有的信号，表示当前显示的场是视频帧的奇场还是偶场。在逐行扫描系统中是不需要场信号的。最后，还有用于所有像素分量的数据时钟信号。

初步了解了 BT.601 建议后，我们来讨论数字视频实现的第二层——ITU 发布的 BT.656 建议。基本上，这个建议是对 BT.601 建议的补充，定义了实现 BT.601 建议所必须的物理接口和数据流。该建议定义了位并行和位串行两种模式，下面仅介绍位并行模式。对于 NTSC 和 PAL 制式信号，该建议规定额定时钟频率为 27MHz，数据行为 8 或 10——取决于广播系统的分辨率。BT.656 建议的最大优点是，数据流中包含了上面讨论的所有同步信号。因此应用只需要实现数据流和时钟信号。BT.656 建议对信号的规定非常直白，H 代表水平同步信号，V 代表垂直同步信号，F 代表场信号。另外，BT.656 既可支持隔行扫描视频，又可

支持逐行扫描视频。

BT.656 建议规定的 NTSC 制式和 PAL 制式的视频帧中，H 位标定了水平消隐区域。H 值为 1 时表示 EAV，即有效视频信号结束；H 值为 0 时表示 SAV，即有效视频信号开始。同样地，当 V 值从 1 变为 0 时，则表明信号从消隐区域变为有效视频区域。对于场信号，也同样是以 F 值的 1、0 变化，表明场 1 和场 2 的转换。

另外，数据流中除了视频数据，还包含控制代码。在如图 4.16 所示的例子中，这个 8 位视频分量的数据流快照中，最前面的几个字节是“FF 00 00”。这是与控制代码相关的前同步码，用于通知终端设备，即将收到控制代码。紧接着，就是“AB”控制代码，负责告知终端设备 H 值、V 值或 F 值是否变化，此外，还有一些用于纠错的校验和位。之后，如果是扫描行的起点，则会有一长串按“80 10 80 10”顺序标定的水平消隐区域。接下来，又是另一个前同步码，告知系统 H 值为 0，即将收到 SAV，即有效视频信号开始。然后，将收到整个视频扫描行——720 个有效像素，等于 1440 字节。最后是 EAV，即有效视频信号结束，开始接收下一个扫描行。

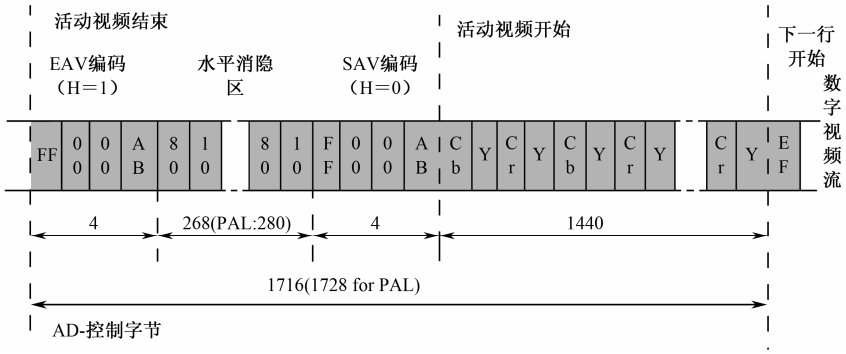


图 4.16　BT.656 数据流格式

下面对视频信号的分析就是针对符合上述标准的数字视频信号展开的。

4.5　基于 ADSP-BF561 的视频采集

基于前面介绍的硬件接口与外设连接的基础知识，本节介绍基于 EZ-KIT 开发板的视频采集实现过程。在硬件平台和硬件连接的基础上，功能的实现离不开软件的支持。在此我们着重介绍设备驱动程序模型和 Blackfin 的系统服务。

对于不同外设，需要得到其驱动程序方能对其进行有效的操作。“设备驱动器模型”提供了一个简单的、干净的、通用的 Blackfin 处理器设备驱动的接口。设备驱动模型的主要目标是创建一个简洁的、高效的、易用的使用接口，通过该接口应用程序可以与设备驱动程序通信。其次，该模型和设备管理软件能够显著地简化设备驱动程序的开发过程，使得新的设备驱动程序的开发变得非常直观。开发板上各模块的驱动程序在 VisualDSP++中已经包含了，因而可以直接使用。

Blackfin 的“系统服务（System Services）”由一组在嵌入式系统中通用的函数组成。每个系统服务集中于一组特定的功能，如直接内存访问 DMA、功率管理 PM、终端控制（IC）等。这些系统服务的全体提供了一组丰富的、预先构建好的且优化过的代码，能够简化用户的软件开发，使得用户能够更快地将其基于 Blackfin 处理器的设计推向市场。

在采用 BF561 的视频采集和显示实现中，采用 Blackfin 的系统服务来简化编程实现。在对 BF533 视频采集实现的介绍中，采用读写外设控制寄存器的低级方式来实现，以进一步揭示其实现原理。

4.5.1　Blackfin 系统服务[39]

Blackfin 系统服务库主要包含以下服务。

（1）中断控制服务：使应用程序更有效地控制、配置事件和中断处理过程。具体功能包括：设置/检测中断优先级到外设的映射；使用标准 C 函数作为中断处理程序；使用嵌套和非嵌套能力将多个中断处理器关联到同一个中断优先级，或从中解关联；检测一个系统中断是否已生效；可移植地对关键区域进行保护或解保护。

（2）功率管理服务：使应用程序能控制 Blackfin 处理器的动态功率管理。主要功能包括：通过函数调用设置内核时钟和系统时钟；设置/检测内部电压调节器的设置；转变处理器的运行模式。

（3）EBIU 服务：提供一组函数来设置 Blackfin 处理器的外部接口，包括 SDRAM 控制器。功能包括：调整 SDRAM 刷新和计时速率到最优值；设置单个总线接口；完成对已知配置（如 EZ-KIT Lite）的单函数设置和启动。

（4）延迟回调服务：使得应用程序能够在高优先级中断服务函数之外得到一般事件的通知。使用延迟回调一般会提高 I/O 的整体处理能力，同时减少中断延迟。功能包括：定义在任一时间点允许的未决回调的数目；定义回调服务在哪个中断优先级执行；创建多个回调服务，每个运行于不同的中断优先级；将回调按相对优先级放入一个回调服务。

（5）DMA 管理服务：提供对 DMA 控制器的服务，允许应用程序规划 DMA 操作，支持线性和二维传输类型。主要功能包括：设置/检测 DMA 通道到外设的映射；配置单个 DMA 通道为输入/输出流，使用循环（自动缓冲）DMA 或基于描述子的 DMA；命令 DMA 管理器在 DMA 完成时发出实时或延迟的回调；对描述子排队，混合 DMA 通道上的线性和二维传输；使能 DMA 管理器自动在描述子链上环回；连续地将数据流入或流出内存、外设；用一个类 C 的 memcpy 函数发起线性和二维内存 DMA 传输。

（6）可编程标志服务：提供对可编程标志的简单接口（GPIO），允许应用程序通过一致的接口来访问和控制可编程标志。功能包括：配置单个标志的方向；设置/清除/切换所有输出标志的值；检测输入标志的值；安装实时或延迟的回调函数，在某标志上满足触发条件时调用。

（7）定时器服务：为应用程序、驱动程序提供一个简单的控制通用、内核、看门狗定时器的机制。主要功能包括：配置并控制各种定时器；安装各种回调函数，在定时器到期

时触发。

（8）端口控制服务：只适用于 ADSP-BF534、ADSP-BF536 和 ADSP-BF537 处理器，用于合适地配置引脚的硬件复用。

（9）设备管理器：设备驱动模型用来控制 ADI 处理器的内部和外部设备。具体功能包括：打开/关闭应用程序使用的设备；配置和控制设备；使用各种数据流方式，通过设备来接收并发送数据。

每个系统服务导出相应的 API 函数。应用程序软件调用这些 API 函数来使用上述功能，每个 API 函数都是使用开发工具箱中的 C 语言运行时间模型的标准调用即可。每个服务的 API 可以使用 C 语言或汇编语言来调用，只要遵从调用 C 语言运行时间模型的惯例和寄存器使用原则即可。

除了应用程序软件要使用系统服务的 API 外，一些系统服务也会调用其他系统服务的 API。大多数情况下，每个服务与其他服务是独立的；然而，允许一个服务使用其他服务的功能可以消除冗余。例如，假设应用程序需要在一个 DMA 描述子完成处理时得到通知，并且应用程序已经请求了延迟回调。在这种情况下，DMA 管理服务会激活延迟回调服务来激活应用程序的回调函数。另一个服务间混合操作的例子是功率管理服务和 EBIU 服务。假设系统有 SDRAM 并且应用程序需要关掉内核和系统时钟以节省功率。当应用程序调用功率管理服务来降低工作频率时，功率服务器自动激活 EBIU 服务，后者调整 SDRAM 刷新率来补偿降低的系统时钟频率。

图 4.17 显示了一组系统服务的集合，以及它们之间的交互。

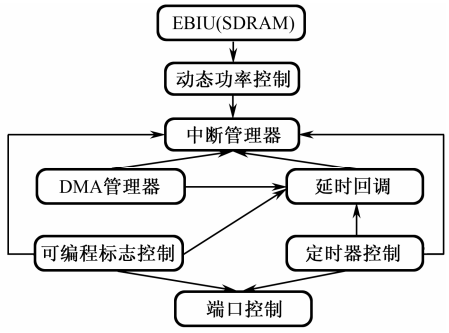


图 4.17　系统服务及 API 接口

应用程序使用单个系统服务或其组合时限制很少，也不必使用所有服务。使用某个服务时也不需要应用该服务控制所有资源。例如，DMA 管理器不需要控制每个 DMA 通道，系统可以配置为 DMA 管理器控制一些通道，让应用程序或其他软件控制其他 DMA 通道。

下面讨论一下各个服务之间的依赖关系。所有的服务（除 EBIU）会激活中断控制服务来管理中断处理。DMA 管理器、回调和功率管理服务都依赖于中断控制服务来为它们管理中断处理。如果应用程序需要功率管理服务来自动调整 SDRAM 定时，就需要使用 EBIU 控制服务来改变 SDRAM 定时参数——当处理器的功率/工作速率改变时。当配置为使用延迟回调时，DMA 管理器利用延迟回调服务的功能来向应用程序提供延迟回调。然而，当配置

为实时回调时，DMA 管理器不使用延迟回调服务。

开发工具箱自动确定这些依赖性，并只将应用程序需要的服务链接到扩展性文件。因为每个服务在系统服务库中是构建为单独的对象文件，可以通过命令连接器去除不需要的对象来进一步减小代码尺寸。

接下来讨论服务的初始化。一些系统服务依赖于其他系统服务，因此存在一个首选的初始化序列。通常最好对所有服务一起进行初始化，一般是当整个系统在被初始化时进行，而不是将不同服务的初始化分布到不同的时刻。以下的初始化序列对大多数应用程序而言是最优化的。序列中应用程序未使用到的任何服务可以简单地从序列中删除。

（1）中断控制服务。

（2）外部总线接口单元服务（EBIU）。

（3）功率管理服务。

（4）端口管理（只用于 ADSP-BF534/536/537 处理器）。

（5）延迟回调服务。

（6）DMA 管理器服务。

（7）可编程标志服务。

（8）定时器服务。

在系统服务终止方面，很多嵌入式系统需要连续运行，执行一个无穷的循环，不会调用到服务的终止函数。不需要终止服务的应用程序可以不用调用终止函数以节省内存。对于那些需要终止服务的应用程序，与初始序列类似，也存在一个首选的服务终止序列。以下的服务终止序列对大多数应用程序是最优的。序列中应用程序未使用的任何服务只需从序列中删除即可。

（1）定时器服务。

（2）可编程标志服务。

（3）DMA 管理服务器。

（4）延迟回调服务。

（5）端口控制（只用于 ADSP-BF534/536/537）。

（6）功率管理服务。

（7）EBIU 服务。

（8）中断控制服务。

4.5.2　Blackfin 设备驱动模型

设备驱动程序为应用程序有效地控制设备提供了一种机制。设备可以是片上或片外硬件设备，或者是可以作为虚拟设备管理的软件模块。设备驱动器的设计原则是将应用程序与硬件（或软件）的细节差别隔离开。这样，被控制的设备驱动程序和设备本身可以被更新或替换，而不会影响应用程序。

ADI 的设备驱动模型的创建为应用程序提供了一个简单的、方便的方法来控制

ADI-DSP 处理器内部和周边的那些设备。它的设计也为新驱动程序的创建提供了方便高效的机制。

下面讨论服务驱动程序与应用程序的接口。设备驱动模型为设备驱动程序提供了一致、简单、规范的应用程序 API。符合该模型的所有设备驱动程序都使用相同的简单接口。大多数设备要么接收数据要么发送数据，有时在处理过程中对数据进行变换。该数据被封装到缓冲区中。缓冲区可能包含少量数据比特，如 UART 类型的设备，每次处理一个字节；可能包含大量数据，如处理 NTSC 帧的视频设备需要大约 1MB 的数据量。应用程序一般向设备提供缓冲区，虽然有些设备可能在无应用程序介入时将缓冲区内容传递给另一个设备。

真正的 API 是符合模型的驱动程序，它包含以下基本函数。

（1）adi_dev_Open()：打开一个设备准备使用。

（2）adi_dev_Close()：关闭一个设备。

（3）adi_dev_Read()：为一个设备提供到达数据的缓冲区。

（4）adi_dev_Write()：为一个设备提供输出数据的缓冲区。

（5）adi_dev_Control()：设置/检测某个设备的控制和状态参数。

与系统服务 API 类似，设备驱动程序 API 也是使用标准 C 语言运行时间模型来调用的。设备驱动 API 可以用 C 语言或汇编语言调用，只要服从 C 语言运行时间模型的调用和寄存器使用习惯。

设备驱动模型将设备驱动程序的功能分为两个部分：设备管理器、物理驱动器。前者是一个软件部件，提供大多数设备驱动程序通用的功能。例如，依赖于应用程序希望设备驱动器的工作方式，应用程序可能会命令一个设备驱动器运行于同步或异步模式。在同步模式下，当应用程序调用 adi_dev_Read()或 or adi_dev_Write() API 函数来从设备读取数据或向设备写入数据时，该 API 函数在所要求的操作完成后才会返回应用程序。在异步模式下，该 API 函数立即返回到应用程序，而数据在幕后进行移动。强制每个硬件驱动器同时提供同步和异步处理的逻辑是浪费的。而设备管理器提供了该功能，避免了每个物理驱动器重复开发该功能。

设备管理器还为应用程序提供了每个设备驱动器的 API。这确保应用程序具有一致的接口，不用关心单个设备的独特性。尽管系统中只有一个设备管理器，但可以有任何数目的物理驱动器。一个物理驱动器是访问、控制某个物理设备的设备驱动器组件。物理驱动器负责该物理设备的真正实现，以及控制寄存器、状态寄存器的操作。所有设备特有的信息包含在物理驱动器中。

如图 4.18 所示，设备驱动模型利用了系统服务的功能。系统中的每个软件组件，不管它是应用程序、RTOS（如果有）、设备管理器，还是物理驱动器，都可以访问并调用系统访问 API。使用该方法的好处很多，除了代码尺寸和数据内存的节省，该方法允许每个软件组件以协作的方式访问系统和处理器的资源。而且，物理驱动器的开发工作极大地减少，因为每个驱动器不需要重复实现已经由设备管理器或系统服务提供的功能。

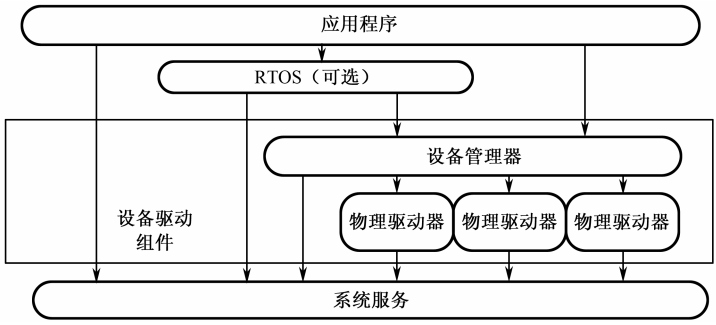


图 4.18　设备管理器结构

在访问单个驱动器之前，设备管理器必须首先初始化。初始化函数是 adi_dev_Init()，由应用程序调用以建立并初始化设备管理器。

尽管设备驱动模型依赖于系统服务，但设备管理器的初始化函数无需依赖于任何系统服务。这样，设备管理器可以在系统服务初始化前或后进行初始化。然而，后续版本的设备管理器初始化函数可能需要系统服务功能。因此，最好是先初始化系统服务，再初始化设备管理器。

设备驱动模型的 API 中包含一个终止函数，应用程序不再需要某设备驱动器时可以调用它来终止它。终止函数 adi_dev_Terminate()会释放该设备管理器和任何打开的物理驱动器使用的资源。许多嵌入式系统以无限循环方式运行，永远不会调用设备管理器的终止函数，这样可以节省程序内存。

作为终止函数处理的一部分，设备管理器将关闭所有打开的物理驱动器。物理驱动器以突发方式关闭。如果需要更安全地关闭设备，应用程序可能会先关闭任何已打开的物理驱动器，然后调用终止函数。

注意，因为对系统服务的依赖，设备管理器的中断函数应该在系统服务中断函数被调用之前被调用，这确保了终止过程中系统服务的可获得性。

设备管理器终止后，必须进行初始化后方能再次访问其功能。

4.5.3　视频采集硬件组成

模拟视频输入方式下的视频采集硬件组成如图 4.19 所示，其中主要是利用了 ADV7183B 来实现视频采集。视频采集软件实现主要利用了 ADV7183B 的设备驱动程序。设备驱动程序为上层应用提供相应的接口，采用独立于任何驱动程序和处理器的标准化 API，并且完全独立运行，而且还可以在不同的 Blackfin 处理器之间进行传送，而应用程序或用户对此则完全不知情，也无需其做出任何更改。

为了在应用程序和驱动程序之间传递各种数据，应用程序负责为设备驱动程序提供缓冲区。在输入方向上，设备驱动程序将接收到的数据写入输入缓冲区，应用程序则从中读

取数据，然后进行后续处理。在输出方向上，应用程序负责将数据写入输出缓冲区，设备驱动程序则从输出缓冲区中读取数据，然后进行视频显示或其他处理。Visual DSP++中包含了 ADV7183B 的设备驱动程序，提供了诸如 ADI_Dev_Read 和 ADI_Dev_write 等函数来进行缓冲区操作。

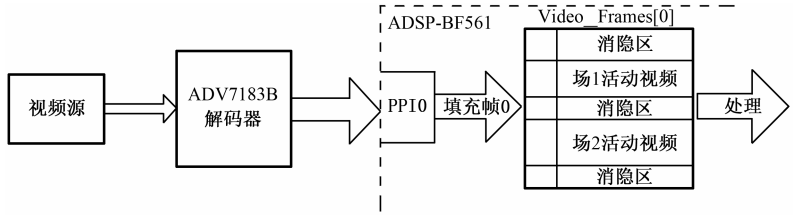


图 4.19　基于 Blackfin 处理器的视频采集数据流

在视频采集过程中，相对于设备驱动程序而言，应用程序发挥的作用微乎其微。应用程序所做的只是对系统服务程序、设备驱动程序初始化。然后，应用程序和设备驱动程序都将完全自己负责管理调用系统服务程序。例如，ADV7183B 设备驱动程序将负责管理 DMA 管理器、中断管理器、定时器控制器及其回调函数。

4.5.4　视频输入数据流

为获取视频信息，需要为 ADV7183B 视频解码芯片配置输入视频源。ADV7183B 既可支持 NTSC 制式，又可支持 PAL 制式，下面的工作采用 NTSC 制式。在 BF561 EZ-KIT 评估板上，已经包含了 ADV7183B 视频解码器，这就免去了外接视频解码器的麻烦，对初学者尽快掌握 DSP 核心开发过程提供了极大的便利。

在视频输入系统中，需要完成视频数据的存储和传输。一般而言，应用会将经 ADV7183B 解码的视频数据填充到 Blackfin 处理器的视频帧中。当视频帧填满后，它会将视频数据传输至另一个外接编码芯片，以执行本地显示，或对数据进行处理及压缩等操作。为了确保不覆盖尚未处理完毕或未显示的像素和视频帧，视频应用基本上都拥有多个缓冲区。如图 4.20 所示，当填充第一个视频帧时，设备驱动程序将在完全填满这个视频帧之后，继续填充下一个视频帧。第一个视频帧填满后，处理器可以放心地处理其中的数据，而不必担心会被新数据或像素改写。这个过程将不断地循环往复。

许多应用都具备多重缓冲区处理特性，即为读写视频提供了多个帧，以避免不必要的处理或改写未处理数据。就设备驱动程序而言，视频应用通常会采用环回链接技术，即设备驱动程序自动地由最后一个缓冲区返回并链接至第一个缓冲区，应用程序不需要去控制这个过程，而是完全由设备驱动程序负责。这是典型的环回过程：填满第一个视频帧后，继续填充第二帧，然后自动环回链接至第一帧。这个过程通常也称“乒乓缓冲”。

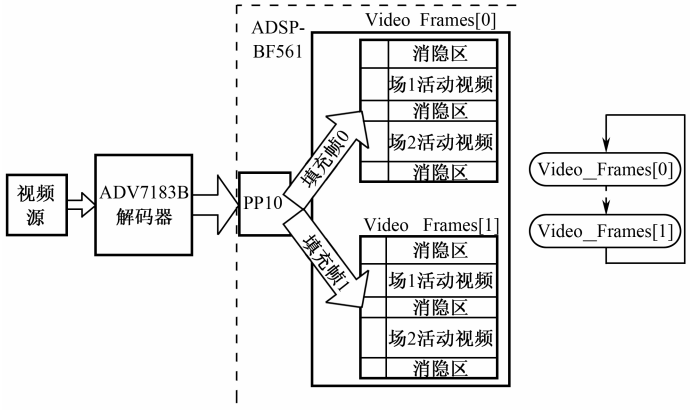


图 4.20　视频采集的多缓冲区处理技术

4.5.5　视频输入实现过程

基于 EZ-Kit 评估板的视频输入系统的开发过程如下。

（1）应用程序初始化系统服务，包括以下内容。

① 针对所需的 SDRAM 初始化 EBIU 模块。

```
ADI_EBIU_COMMAND_PAIR ezkit_sdram[] = {
{ ADI_EBIU_CMD_SET_SDRAM_BANK_SIZE,(void*)&bank_size },
{ ADI_EBIU_CMD_SET_SDRAM_MODULE,
(void*)ADI_EBIU_SDRAM_MODULE_MT48LC16M16A2_75 },
{ ADI_EBIU_CMD_END, 0}
adi_ebiu_Init( ezkit_sdram, DO_NOT_CHANGE_MMR_SETTINGS );
```

其中 ezkit_sdram 的参数值应该按照内存模块的数据表进行设置，本例中对应的 ADSP-BF561 EZ-KIT 使用两个 Micron MT48LC16M16A2-75 内存模块（4Meg×16×4），共有 64MB 的内存空间。

② 初始化功率管理模块。

```
ADI_PWR_COMMAND_PAIR ezkit_power[] = { /*使用 BF561 EZ-KIT*/
     { ADI_PWR_CMD_SET_EZKIT,    ADI_PWR_EZKIT_BF561_600MHZ},
     { ADI_PWR_CMD_END, 0} };
adi_pwr_Init( ezkit_power );
adi_pwr_SetFreq( 0, 130, ADI_PWR_DF_OFF);    /*尝试最高频率*/
*pEBIU_SDGCTL |= 2; /*打开时钟输出以检查 SCLK（写寄存器方式）*/
```

③ 初始化中断管理器，同时关联起异常和硬件错误中断。

```
 adi_int_Init(NULL, 0, &ResponseCount, NULL);
 adi_int_CECHook(3, ExceptionHandler, NULL, FALSE);
```

```
adi_int_CECHook(5, HWErrorHandler, NULL, FALSE);
```

其中 adi_int_Init()用来初始化中断管理器，为中断管理器留出内存并初始化该内存，接着初始化中断管理器中的各个表格和向量。对每个内核它只能调用一次，同时每个内核应该分配单独的内存区域。

adi_int_CECHook()指示中断管理器将给定的中断处理器插入给定 IVG 的中断处理器链中。这样系统就可以通过中断管理器内置的中断服务程序（ISR）来正确地服务中断。ISR 接下来就会激活调用者提供的中断处理器。对给定的 IVG 级别，第一次调用 adi_int_CECHook 时，中断管理器注册其内置 IVG 中断服务函数，并将所提供的中断处理器建立为主中断处理器，接下来对该函数的调用（对同一个 IVG 级别）创建次中断处理器链。当该 IVG 级中断触发时，首先调用主中断处理器，然后再顺序调用各个次中断处理器。

④ 初始化 DMA 管理器。

```
adi_dma_Init(DMAMgrData, sizeof(DMAMgrData), &ResponseCount,
             &DMAManagerHandle, NULL);
```

前两个参数是 DMA 可以使用的内存指针及内存尺寸，定义如下：

u8 DMAMgrData[ADI_DMA_BASE_MEMORY + (ADI_DMA_CHANNEL_MEMORY * 1)];第三个参数返回所支持的最大通道数，第四个参数返回 DMA 管理器的句柄。

⑤ 初始化设备管理器（在系统服务之后）。

```
adi_dev_Init(DevMgrData, sizeof(DevMgrData), &ResponseCount,
             &DeviceManagerHandle, NULL);
```

adi_dev_Init()创建一个设备管理器，并为设备管理器初始化内存。前两个参数是设备管理器可使用的内存地址及尺寸，第三个参数返回可同时打开的设备最大数目，第四个参数返回设备管理器句柄。

（2）为输入视频数据设置循环缓冲区，即“乒乓缓冲”（PingBuffer[NUM_BUFFERS]），这里 NUM_BUFFERS 为数组元素个数，数组元素数据类型是 ADI_DEV_2D_BUFFER，定义如下：

```
typedef struct adi_dev_2d_buffer {/*2D缓冲区*/
    char Reserved[ADI_DEV_RESERVED_SIZE]; /* 为物理驱动器保留 */
    void *Data;                    /* 指向数据 */
    u32 ElementWidth;              /* 数据元素宽度（字节数） */
    u32 XCount;                    /* 2D缓冲区XCOUNT值 */
    s32 XModify;                   /* 2D缓冲区XMODIFY值 */
    u32 YCount;                    /* 2D缓冲区YCOUNT值 */
    s32 YModify;                   /* 2D缓冲区YMODIFY值 */
    void *CallbackParameter;       /* 回调标志/pArg值 */
    volatile u32 ProcessedFlag;    /* 已处理标志 */
    u32 ProcessedElementCount;     /* 已读入/出元素数 */
    struct adi_dev_2d_buffer *pNext; /* 下一个缓冲区 */
```

```
    void *pAdditionalInfo;        /* 设备特定附加信息指针 */
} ADI_DEV_2D_BUFFER;
```

下述代码将缓冲区设置为循环缓冲区，具体数值需根据具体应用来确定。最后一行代码将最后一个缓冲区的 pNext 设为 NULL，通知它需要环回到首个缓冲区。

```
for (i = 0; i < NUM_BUFFERS; i++) {
     PingBuffer[i].Data = PingFrame;     /*需要指向对应的缓冲区地址*/
     PingBuffer[i].ElementWidth = 4;     /*视频元素字节数*/
     PingBuffer[i].XCount = 320;
     PingBuffer[i].XModify = 4;
     PingBuffer[i].YCount = 480;
     PingBuffer[i].YModify = 20;
     PingBuffer[i].CallbackParameter = NULL;      /*可设特定值以区分缓冲区*/
     PingBuffer[i].pNext = &PingBuffer[i + 1];  /*指向下一个缓冲区*/
}
PingBuffer[NUM_BUFFERS - 1].CallbackParameter = &PingBuffer[0];
PingBuffer[NUM_BUFFERS - 1].pNext = NULL;    /*这是循环缓冲区最后一个*/
```

其中，PingBuffer[]对应的缓冲区指向不同的视频帧，如最简单情况下包括视频帧 0 和视频帧 1（Video_Frame[0]和 Video_Frame[1]）。此外还定义了视频元素的宽度。为优化利用 Blackfin 处理器的带宽，往往采用打包技术，对 32 位宽的数据元素进行打包。此外还要设置帧的大小，并通知设备驱动程序将接收多少视频信号。上例中视频帧尺寸为 640×480，由于采用打包技术，XCOUNT 数值应设置为 320。

可以通过设置 PingBuffer[i]的回调参数 CallbackParameter。在双缓冲情况下需要两个回调指示器：第一个是“1”，表明第一个视频帧已经填满；而另一个回调指示器则是“2”，表明第二个视频帧已经填满。

（3）打开 PPI 驱动器：

```
adi_dev_Open( DeviceManagerHandle, & ADIPPIEntryPoint, PPI0, NULL,
              &DriverHandle, ADI_DEV_DIRECTION_INBOUND,
              DMAManagerHandle, DCBManagerHandle, Callback );
```

adi_dev_Open()用来打开设备。它首先找到一个空闲的 ADI_DEV_DEVICE 数据结构，然后用描述设备的相关信息设置该结构，接下来设备管理器调用物理驱动器的 adi_pdd_Open 函数。物理驱动器开始执行，打开它控制的设备。

第一个参数是设备管理器句柄 DeviceManagerHandle，来自前面对 adi_dev_Init 的调用，其类型是 ADI_DEV_MANAGER_HANDLE；第二个参数是 PDD 入口点，后面具体介绍；第三个参数是设备号，PPI0 的值是 0；第四个参数是客户端句柄指针；第五个参数是获得的设备句柄指针；第六个参数是数据方向，ADI_DEV_DIRECTION_INBOUND 代表读入，ADI_DEV_DIRECTION_OUTBOUND 代表写出，ADI_DEV_DIRECTION_BIDIRECTIONAL 代表双向；第七个参数是 DMA 管理器句柄 DMAManagerHandle，是从上面 adi_dma_Init 调

用获得的；第八个参数是回调管理器句柄；第九个参数是客户端回调函数 Callback，原型是 void funcName(void *AppHandle, u32 Event,void *pArg)。

ADIPPIEntryPoint 是文件 adi_ppi.c 中定义的全局变量，如下所示：

```
ADI_DEV_PDD_ENTRY_POINT ADIPPIEntryPoint = {
     adi_pdd_Open,   /*打开 PPI 设备*/
     adi_pdd_Close,  /*关闭 PPI 设备*/
     adi_pdd_Read,   /*不使用，因为 PPI 使用 DMA*/
     adi_pdd_Write,  /*不使用，因为 PPI 使用 DMA*/
     adi_pdd_Control /*配置 PPI 设备*/
};
```

它给出了执行 PPI 接口功能所需要的实现函数。

在客户回调函数 Callback 中，视频处理系统根据 Event 的值进行不同的处理，主要处理 ADI_DEV_EVENT_BUFFER_PROCESSED 事件，此时 pArg 是指向缓冲区的指针，其类型是(ADI_DEV_BUFFER *)，可以指向一维缓冲区、二维缓冲区、循环缓冲区、一维序列缓冲区等。在这里，应用程序就可以开始对缓冲区中的视频数据进行处理了。

打开 PPI 设备后，需要对其进行配置。以下是 PPI 配置数据表，它指定了数据流方式、控制寄存器值、帧分辨率等，并以 ADI_DEV_CMD_END 为结尾。

```
ADI_DEV_CMD_VALUE_PAIR ConfigurationTable [] = {/*PPI 驱动配置值表*/
     { ADI_DEV_CMD_SET_DATAFLOW_METHOD,
                        (void *) ADI_DEV_MODE_CHAINED_LOOPBACK },
     { ADI_PPI_CMD_SET_CONTROL_REG,          (void *)0x592C      },
     { ADI_PPI_CMD_SET_DELAY_COUNT_REG,      (void *)0       },
     { ADI_PPI_CMD_SET_TRANSFER_COUNT_REG,   (void *)(640-1)     },
     { ADI_PPI_CMD_SET_LINES_PER_FRAME_REG, (void *)480          },
     { ADI_DEV_CMD_SET_STREAMING,            (void *)TRUE    },
     { ADI_DEV_CMD_END,                      NULL                }   };
```

利用表中的属性值对列表、对 PPI 进行配置：

```
adi_dev_Control(DriverHandle, ADI_DEV_CMD_TABLE, ConfigurationTable);
```

然后将循环缓冲链 PingBuffer[]提供给设备使用：

```
adi_dev_Read(DriverHandle, ADI_DEV_2D, (ADI_DEV_BUFFER *)PingBuffer);
```

参数分别是驱动器句柄、缓冲区类型和缓冲区指针，ADI_DEV_2D 代表二维缓冲区。

（4）启动数据流：

```
adi_dev_Control(DriverHandle,ADI_DEV_CMD_SET_DATAFLOW,(void*)TRUE);
```

第二个参数是 ADI_DEV_CMD_SET_DATAFLOW 时表明要启动/停止数据流，第三个参数为 TRUE 代表启动，反之代表停止。

完成以上初始化和启动工作后，系统开始接收到视频源的视频信号。视频信号将通过 ADV7183A 转换为符合 BT.656 格式的数字信号进入 Blackfin 处理器，开始填充第一个视频

帧。数据传输过程都是由设备驱动程序完成的。由于采用了环回链接，当视频帧 0 填满时，数据流将自动转而填充视频帧 1。这样，设备驱动程序就可以来回利用视频帧 0 和视频帧 1 了。在用户自定义回调函数 Callback 中，如果触发事件是 ADI_DEV_EVENT_BUFFER_PROCESSED，就获得了输入的视频数据，用户可以根据需要对数据进行各种处理了。

视频输入乒乓缓冲工作示意图如图 4.21 所示。

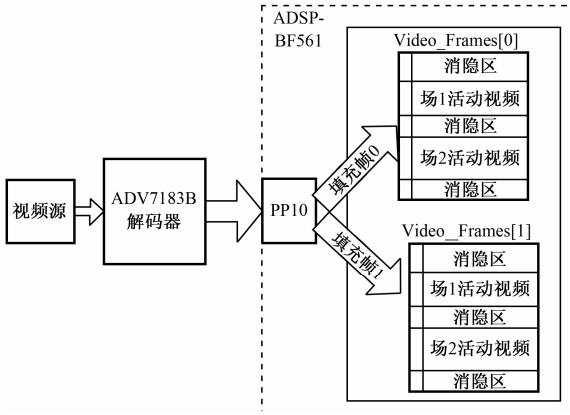


图 4.21　视频输入乒乓缓冲工作示意图

在此总结一下视频捕获的实现过程。首先，初始化系统服务和设备驱动，包括 EBUI、功率控制、中断控制管理器、DMA 管理器等；接着，为接收输入数据流，必须为设备驱动程序配置缓冲区，如采用循环缓冲区；然后打开设备，设置设备参数，包括视频尺寸、回调函数等；最后，启动数据流，当视频帧数据填满后，Blackfin 处理器将接收到视频源的视频信号，并在用户回调函数中进行处理。执行回调期间，回调函数一般自己会为驱动程序提供一个额外的缓冲区，以便填充数据。

实现过程中应用程序与驱动程序间的调用关系如图 4.22 所示。在应用程序完成对驱动程序的初始化和启动后，驱动程序独立执行，并且在数据填满后调用应用程序中的回调函数。

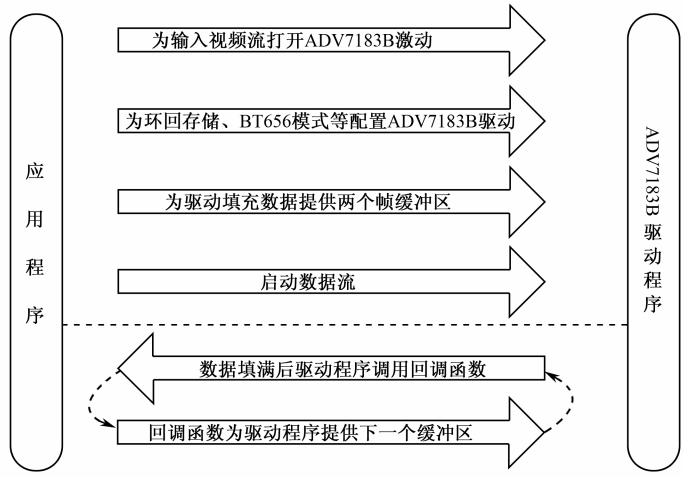


图 4.22　基于 ADV7183B 的视频采集编程过程

4.6 基于 Blackfin 处理器的视频输出

4.6.1 视频输出数据流

视频输出与视频输入非常相似，主要使用 ADV7179 器件和 Blackfin BF561 EZ-KIT 评估板进行开发。系统可以在一台本地连接的电视机上显示视频编码器 ADV7179 输出的 NTSC 制式视频信号，如图 4.23 所示。

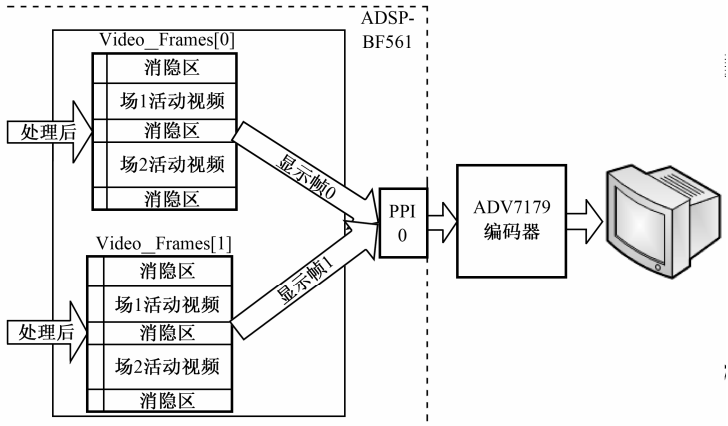


图 4.23 基于 Blackfin 的视频输出

ADV7179 位于 BF561 EZ-KIT 评估板上，Visual DSP++提供了 ADV7179 的设备驱动程序。该步骤将已经处理的数据或帧数据填充到视频帧中，设备驱动程序会将这些视频帧数据经 PPI 接口传输至 ADV7179，并由其将这些数据编码为标准制式的视频信号，再在本地显示器上显示。

与基于 ADV7183A 的视频输入一样，输出应用也要使用乒乓缓冲，以防止新输出的视频帧数据改写正在显示的信号。与 ADV7183A 设备驱动程序一样，这里也采用了循环缓冲区模式，设备驱动程序将自主控制缓冲区，确保数据源源不断地传输。完成传输并显示第一个视频帧 0 后，将传输和显示下一个视频帧——视频帧 1；视频帧 1 显示完毕后，设备驱动程序将在回调过程中加载视频帧 0。

4.6.2 视频显示实现过程

基于 EZ-KIT 评估板的视频输出系统的开发过程与视频输入大体相同，也是分为初始化系统服务程序、准备视频缓冲区、打开 PPI 设备并进行设置、启动视频流这些部分，但存在以下不同之处。

（1）为了显示的同步，在 PWM 模式下，需要设置定时器来控制 HSYNC 和 VSYNC。

本例中 Timer0、Timer8、Timer9 分别用于 PPI_CLK、HSYNC 和 VSYNC。

首先打开定时器：

```
adi_tmr_Open(ADI_TMR_GP_TIMER_X)。
```

然后定时器属性对表对定时器进行编程：

```
adi_tmr_GPControl(ADI_TMR_GP_TIMER_X, ADI_TMR_GP_CMD_TABLE,
                  TimerXConfigurationTable);
```

其中的 TimerXConfigurationTable 中包含了定时器的参数值，例如：

```
ADI_TMR_GP_CMD_VALUE_PAIR Timer8ConfigurationTable [] = {
     { ADI_TMR_GP_CMD_SET_TIMER_MODE,        (void *)0x01        },
     { ADI_TMR_GP_CMD_SET_PULSE_HI,          (void *)FALSE       },
     { ADI_TMR_GP_CMD_SET_COUNT_METHOD,      (void *)TRUE        },
     { ADI_TMR_GP_CMD_SET_INPUT_SELECT,      (void *)TRUE        },
     { ADI_TMR_GP_CMD_SET_CLOCK_SELECT,      (void *)TRUE        },
     { ADI_TMR_GP_CMD_RUN_DURING_EMULATION, (void *)TRUE        },
     { ADI_TMR_GP_CMD_SET_PERIOD,            (void *)996         },
     { ADI_TMR_GP_CMD_SET_WIDTH,             (void *)356         },
     { ADI_TMR_GP_CMD_END,                   NULL                }};
```

就给出了定时器的周期、脉冲宽度、高/低电平有效等参数的值。

注意：此处只是对定时器进行了设置，还没有使能它们。接下来首先要启动 Timer0 以生成时钟信号：

```
adi_tmr_GPGroupEnable(ADI_TMR_GP_TIMER_0, TRUE);
```

而在启动视频数据流之后，需要使能 Timer8 和 Timer9 以生成 PPI 的同步信号：

```
adi_tmr_GPGroupEnable(ADI_TMR_GP_TIMER_8 | ADI_TMR_GP_TIMER_9,TRUE);
```

（2）在打开 PPI 驱动时，数据方向改为 ADI_DEV_DIRECTION_OUTBOUND。

```
adi_dev_Open(DeviceManagerHandle, &ADIPPIEntryPoint, PPI0, NULL,
             &DriverHandle, ADI_DEV_DIRECTION_OUTBOUND,
             DMAManagerHandle, DCBManagerHandle, Callback);
```

同时 PPI 的配置信息也发生了变化：

```
ADI_DEV_CMD_VALUE_PAIR ConfigurationTable [] = {
{ ADI_DEV_CMD_SET_DATAFLOW_METHOD,
                        (void *)ADI_DEV_MODE_CHAINED_LOOPBACK },
{ ADI_PPI_CMD_SET_CONTROL_REG, (void *)0x011C },/*设置 PPI 控制寄存器*/
{ ADI_PPI_CMD_SET_FS_INVERT, (void *)TRUE },   /*翻转帧同步极性*/
{ ADI_PPI_CMD_SET_CLK_INVERT, (void *)TRUE },  /*翻转 PPI 时钟极性*/
{ ADI_PPI_CMD_SET_DATA_LENGTH, (void *)15 },   /*数据长度*/
{ ADI_PPI_CMD_SET_SKIP_EVEN_ODD, (void *)FALSE }, /*不跳过偶数元素*/
{ ADI_PPI_CMD_SET_SKIP_ENABLE, (void *)FALSE },   /*控制是否跳过*/
```

```
{ ADI_PPI_CMD_SET_PACK_ENABLE, (void *)FALSE },/*是否打包*/
{ ADI_PPI_CMD_SET_ACTIVE_FIELD_SELECT, (void *)FALSE },/*选择活动场*/
{ ADI_PPI_CMD_SET_PORT_DIRECTION, (void *)TRUE }, /*设置端口方向*/
{ ADI_PPI_CMD_SET_TRIPLE_FRAME_SYNC, (void *)FALSE },/*选 3 帧同步*/
{ ADI_PPI_CMD_SET_DELAY_COUNT_REG, (void *)355 }, /*设置延迟计数*/
{ ADI_PPI_CMD_SET_TRANSFER_COUNT_REG, (void *)639 }, /*设置传输计数*/
{ ADI_DEV_CMD_SET_STREAMING, (void *)TRUE },       /*设置流模式*/
{ ADI_DEV_CMD_END, NULL } };
```

（3）为 PPI 提供输出缓冲区是调用 adi_dev_Write 函数：

```
adi_dev_Write(DriverHandle, ADI_DEV_2D, (ADI_DEV_BUFFER *)PingBuffer)
```

（4）在用户回调函数 Callback 中，事件 ADI_DEV_EVENT_BUFFER_PROCESSED 意味着缓冲区数据已经成功写出，参数 pArg 指向的缓冲区又成为可写的了。

简单总结一下视频显示的实现过程。首先，启动系统服务和设备驱动，设置定时器 Timer0、Timer8 和 Timer9，接着启动 Time0 生成 PPI_CLK 信号；然后，打开 PPI 驱动并设置视频输出的模式和属性，为驱动程序提供循环缓冲区，并发起数据流；最后还需要启动 Timer8 和 Timer9 以输出帧同步信号。

图 4.24 显示了应用程序与 ADV7179 驱动程序的交互过程，应用程序完成对驱动程序的设置后启动数据流，应用程序向缓冲区写入视频数据，帧缓冲区数据发送给驱动程序继续显示。当上一个满载缓冲区中的视频帧显示完毕后，设备驱动程序将生成一个回调给应用程序。然后，应用程序将确认回调，并自主选择向这个视频帧填充新的数据，或者不做任何处理。设备驱动程序在生成回调命令后，将开始显示循环缓冲区中下一个缓冲区中的视频帧。

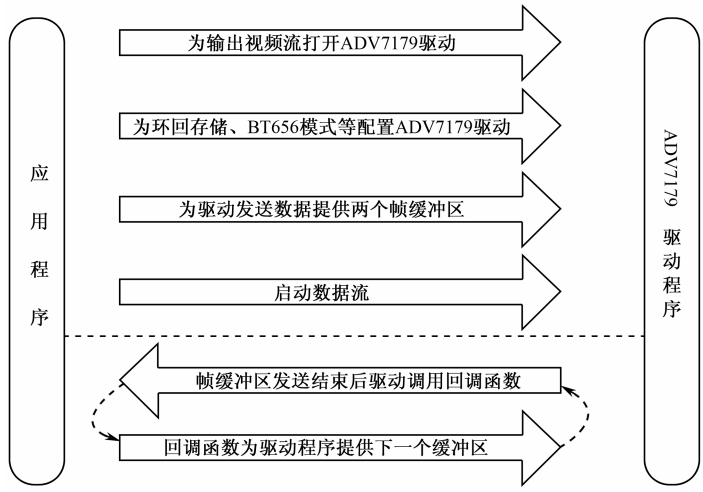


图 4.24　基于 ADV7179 的视频显示实现过程

4.6.3　基于 Blackfin 处理器的视频传输

视频输入与输出数据流示意图如图 4.25 所示。

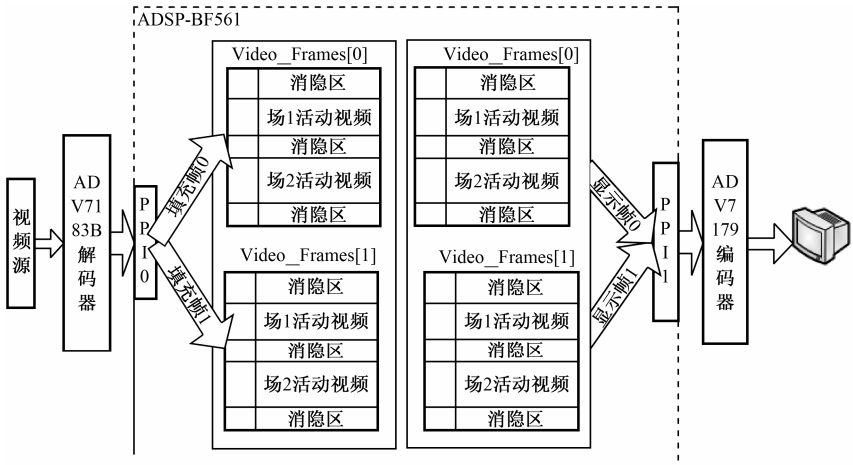


图 4.25　视频输入与输出数据流示意图

如前所述，为实现视频的输入和输出，首先需要安装 ADV7183B 和 ADV7179 的设备驱动程序。前者负责将视频源输出的视频信号经 ADV7183B 解码后填充到 Blackfin 处理器的两个视频帧中，后者负责将 Blackfin 处理器中两个满载的视频帧经 ADV7179 编码后在本地显示器上显示。此处可以考虑将图 4.25 中的缓冲区进行重用，构成一个简单的视频传输系统，如图 4.26 所示。

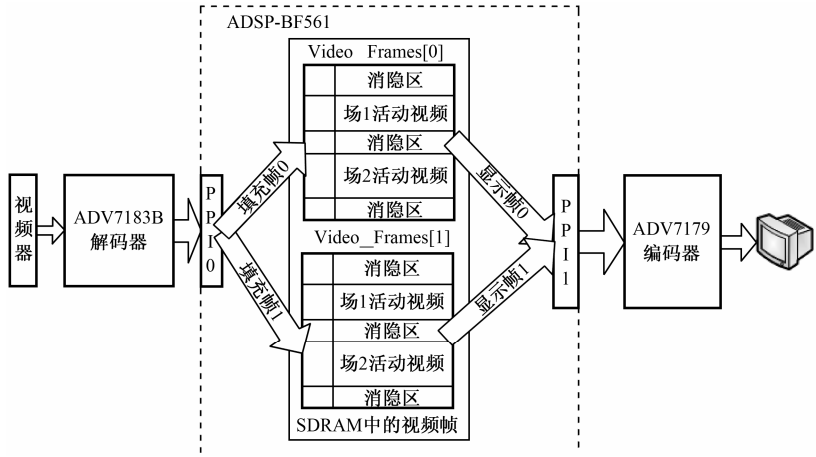


图 4.26　基于 Blackfin 处理器的视频传输系统组成

由于两个驱动程序之间相互独立，而且可以共享相同的视频帧缓冲区，这样就实现了一个真正“简单的”视频传输系统。具体地，ADV7183B 设备驱动程序负责接收视频帧信

号，并开始填充视频帧 0。当视频帧 0 填满后，ADV7179 设备驱动程序就开始输出这个视频帧信号，并在本地显示器上显示。与此同时，ADV7183B 设备驱动程序将开始填充这个环回链接中的下一个视频帧。该过程将周而复始，通过来回利用这两个视频帧，实现一个简单的视频传输。

4.7 基于 ADSP-BF533 的视频采集与显示

以上介绍的是采用系统服务实现的基于 ADSP-BF561 的视频采集与显示。系统服务方便用户快速高效地开发应用程序，但是它将底层实现封装了起来，不容易理解具体的实现过程。为此，接下来基于 ADSP-BF533 介绍一下上述内容的具体实现。

基于 BF533 的实现从外设、PPI 接口、实现过程都非常类似，但它是单核的、只有一个 PPI 接口，所以不能实现视频采集和显示的并行运行。在下面的例子中，主要通过写设备控制寄存器的方式来实现硬件的初始化和配置、中断控制的管理等任务，相关代码直观、高效，体现了与硬件的紧密联系。

下面我们以视频采集为例，介绍其实现过程。

系统初始化工作包括初始化系统的工作频率、EBUI 接口、Flash GPIO、视频解码器 ADV7183 等，以及中断服务、SDRAM、DMA 控制器和 PPI 接口。在 DMA 和 PPI 初始化中，使能 DMA 和 PPI 并打开，这样视频数据流就会通过 PPI 端口经 DMA 控制器传输到内存中，当数据填满后就会触发中断服务初始化设置的 DMA0 中断服务函数 ISR，该 ISR 就是应用程序处理视频数据的入口。下面介绍一下其中的主要实现代码，完整的代码请参考 VisualDSP++安装路径下的例子项目 BF533_EZ_KIT_Video_Receiver_C。

4.7.1 硬件平台初始化部分

（1）SDRAM 设置（在 Init_SDRAM 函数中实现）。

pEBIU_SDSTAT 是 SDRAMN 状态寄存器地址，其 SDRS（=0x0008）位代表是否在下一次访问时上电。如果该位为 1，则设置以下 3 个寄存器：

```
*pEBIU_SDRRC = 0x00000817;      // SDRAM 刷新率控制寄存器
*pEBIU_SDBCTL = 0x00000013;     // SDRAM 内存块控制寄存器
*pEBIU_SDGCTL = 0x0091998d;     // SDRAM 内存全局控制寄存器
```

然后执行 ssync()以使上述设置生效。

（2）初始化 Flash。要使用 Flash 的 GPIO，必须初始化它。

```
*pFlashA_PortA_Out = 0x0;       /*清除数据寄存器*/
*pFlashA_PortA_Dir = 0xFFFF;    /*设置方向都为输出*/
```

（3）将 ADV7183 时钟连接到 PPI 和 ADV7183 重置引脚（函数 Init_ADV 中）。Flash 的 GPIO 要连接到 ADV7183 重置引脚及驱动 PPI 时钟的开关。先将 ADV7183 的重置失效，

然后将 ADV7183 时钟 LLC1 连接到 Blackfin 的 PPI 时钟输入。

```
tempReg = *pFlashA_PortA_Out;        /*设置 FlashA 的 PortA*/
*pFlashA_PortA_Out = tempReg | RST_7183 | PPICLK_ADV7183_SELECT;
```

其中 RST_7183（0x8）是 FlashA 的 PortA 的解码器重置位，PPICLK_ADV7183_SELECT（0x10）则是解码器时钟连接到 PPI 对应的位。

```
然后设置 Blackfin 的 PF 寄存器，ADV7183_OE 的值是 0x4（第二位）：
tempReg = *pFIO_INEN;                /*Flag 输入使能寄存器*/
*pFIO_INEN = tempReg | ADV7183_OE;
Blackfin PF2 引脚必须设置为输出：
tempReg = *pFIO_DIR;                 /*外设 Flag 方向寄存器*/
*pFIO_DIR = tempReg | ADV7183_OE;
设置 Blackfin 引脚 PF2 为输出，使能 ADV7183 数据总线：
tempReg = *pFIO_FLAG_C;              /*外设中断 Flag 寄存器*/
*pFIO_FLAG_C = tempReg | ADV7183_OE;
```

4.7.2　初始化中断服务

（1）将 DMA0 的 PPI 中断映射到 IVG8：

```
*pSIC_IAR0 = *pSIC_IAR0 & 0xffffffff | 0x00000000;
*pSIC_IAR1 = *pSIC_IAR1 & 0xffffffff | 0x00000001;
*pSIC_IAR2 = *pSIC_IAR2 & 0xffffffff | 0x00000000;
```

（2）注册 DMA0 的 PPI 中断服务程序（ISR）分配给中断向量 8：

```
register_handler(ik_ivg8, DMA0_PPI_ISR);
```

其中 DMA0_PPI_ISR 是应用程序定义的 ISR，定义方法如下：

```
EX_INTERRUPT_HANDLER(DMA0_PPI_ISR){
 /*最简单的情况下，停止 DMA0 的中断请求*/
 *pDMA0_IRQ_STATUS = 0x1;
 /*一般情况下，此时已获得视频数据，可以开始数据的处理过程了*/
}
```

（3）打开系统中断掩码：

```
*pSIC_IMASK=0x00000100;
```

这里只有 DMA0 中断是使能的，其他的中断都是被禁止的。

4.7.3　初始化 DMA

初始化 DMA 需要配置如下寄存器的数值：

```
*pDMA0_START_ADDR = 0x0;          /*DMA 目的地址寄存器*/
*pDMA0_X_COUNT = 720;             /*X_COUNT 值*/
```

```
*pDMA0_X_MODIFY = 0x2;          /*X_MODIFY 值，2 代表元素按 16 位传输*/
*pDMA0_Y_COUNT = 525;           /*Y_COUNT 值（帧长度）*/
*pDMA0_Y_MODIFY = 0x2;          /*Y_MODIFY 值*/
*pDMA0_PERIPHERAL_MAP = 0x0;    /*使用 PPI 外设*/
```

最后设置 DMA0 配置寄存器，内容包括：使能 DMA 通道、使能数据中断、通道方向为写、字长为 16 比特、二维 DMA 模式、开始前丢弃 DMA FIFO 内容、使能数据中断。

```
*pDMA0_CONFIG = DMAEN | DI_EN | WNR | WDSIZE_16| DMA2D | RESTART | DI_EN;
```

这时 DMA0 通道就被打开了。

4.7.4 初始化 PPI

PPI 设置为每帧接收 525 行视频数据：

```
*pPPI_FRAME = 525;
```

配置 PPI 控制寄存器，内容包括：PPI 端口使能、选择 PPI 活动场、采用 PPI 打包模式、8 位数据总线、不做任何翻转、不做跳过处理。

```
*pPPI_CONTROL = PORT_EN | FLD_SEL | PACK_EN | DLEN_8 ;
```

此时 PPI 端口也打开了，数据就开始通过视频源、解码器、PPI 端口、DMA 控制器传输到（*pDMA0_START_ADDR）指向的位置，缓冲区填满后触发 DMA0 中断，进入上述定义的中断处理函数 DMA0_PPI_ISR 中，此时即可对视频数据进行后续处理。

可以看出，对系统的初始化和配置都是通过读写各个外设对应的寄存器位来实现的，因此代码与硬件实现紧密相关。中断管理则是通过设置系统中断分配寄存器、系统中断掩码寄存器，并向事件向量表 EVT 中注册 ISR 来实现的，对于不同事件导致的中断响应，需要通过访问外设状态寄存器来进一步确定是由哪一个（些）系统中断触发的。

视频输出的实现过程与 BF561 的实现也很类似，其中用到的寄存器与上述 BF533 的视频采集过程是一致的，只是各个寄存器中的内容存在很明显的区别，具体细节请参考相关的代码，本节就不再展开讨论了。

4.8 视频采集回放及编码系统的实现

在实现视频采集与显示功能的基础上，还可以用软件对采集到的视频信号进行处理或压缩。这可以是 Soble 边缘检测（如用于车道检测），也可以是 MJPEG（Motion JPEG）或 MPEG 编码。ADI 提供了 MJPEG 的实现，下面将讨论其实现细节。

ADI 的 MJPEG SDK 是单独的编解码器库[40]。编解码在 Blackfin SDK 中以库的形式提供，示例程序的其他部分以源代码形式提供。MJPEG 对输入的 4∶2∶0 格式（亮度和色度使用单独的缓冲区）的视频帧进行压缩、编码，结果保存到压缩数据缓冲区。ADV7183 输出的是 YCrCb 格式，系统得到的是 4∶2∶2 隔行格式的视频数据。此时用户一般在回调函数中启动内存 DMA，来将 4∶2∶2 格式转换为 4∶2∶0 格式，同时对 Cr 和 Cb 分量进行下采样。这

样 MJPEG 编码器就可以直接对 4:2:0 格式的亮度、色度数据进行编码了。

MJPEG 视频数据格式转换过程如图 4.27 所示。

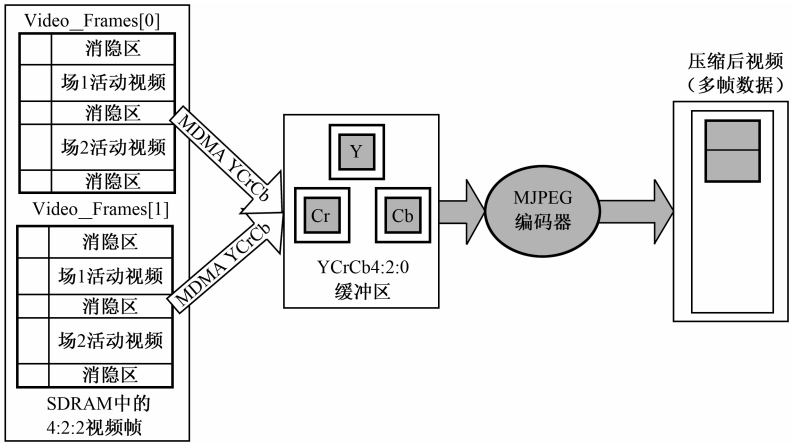


图 4.27　MJPEG 视频数据格式转换过程

MJPEG 编码的实现代码在函数 MJPEG_encode()中。使用 JPEG 软件编码器的第一步是配置它，配置内容包括帧宽度、帧高度和质量因子等。表 4.1 显示了部分配置选项及对应的性能需求，包括帧尺寸、帧速率、质量因子、压缩率、处理每像素所需时钟数等。例如，设置帧宽度为 Input_Width 的代码为：

```
JPEG_Param_CONFIG( & lImageParam, JPEG_FRAME_WIDTH,
                   Input_Width);
```

第一个参数类型为 tJpegParam，包含了 JPEG 图像的属性、编解码类型（基线、无损等）、输入缓冲区指针、输出缓冲区指针、编码质量指标等。第二个参数还可以是 JPEG_FRAME_HEIGHT、JPEG_IMAGEFORMAT、JPEG_QUALITYFACTOR、JPEG_ENCODINGMODE 等，分别用于设置不同的参数。

表 4.1　JPEG 编码器配置选项表

水　平	垂　直	帧/s	质 量 因 子	压　缩　率	时钟数/像素	所需 MHz
176	144	30	60	10.3	53.89	41
176	143	30	40	13.1	49.08	37
640	480	30	60	14.0	43.68	403
640	480	30	40	19.2	38.7	357

除了配置 JPEG，还要初始化或分配一个输出流，以便将数据输出到本地缓冲区或通过 USB 等接口传输到 PC 文件中。

```
StreamBuffer_Obj = JPEG_MemAlloc_NEW(3 * Input_Width *
                                     Input_Height,1,MEM_TYPE_DATA);
StreamBuffer = (uint8*)JPEG_MemAlloc_ADDRESS(StreamBuffer_Obj);
```

最后，初始化 JPEG 编码器，将新缓冲区指针传给它，并创建 JPEG 编码器实例。

```
JPEG_Param_CONFIG(&lImageParam, JPEG_POINTER_OUTPUT,
                                    (int)StreamBuffer);
lJpegEnc = JPEG_Encoder_NEW(&lImageParam);
```

如果是本地保存为 AVI 文件，需要建立 MJPEG 编码器。

（1）创建 AVI 输出文件：

```
MJPEG_AVI_OpenFileWrite( & lStreamFileOutHandle, (uint8*) input_file,
                                    &lImageParam, FrameRate);
```

（2）根据 JPEG 参数设置流参数：

```
MJPEG_AVI_SetStreamParams (&lStreamInfo, &lImageParam, FrameRate);
```

（3）创建输出流：

```
MJPEG_AVI_OpenStreamWrite(lStreamFileOutHandle, &lStreamHandle,
                                    &lStreamInfo);
```

（4）由 JPEG 参数复制帧格式：

```
MJPEG_AVI_CopyParams (&lBitMapInfo, &lImageParam);
```

（5）由 JPEG 参数设置流格式：

```
MJPEG_AVI_SetFormat(lStreamHandle, 0, (uint8 *)&lBitMapInfo,
                                    lFormatLength);
```

然后，初始化 MDMA 输入描述子链，以接收来自 PPI 的视频输入，具体代码参见函数 InitDescriptorChains_Input()。其中，描述子 0 和 6 分别对应 Y 分量第一部分的源和目标，1 和 7 分别对应 U 分量第一部分的源和目标；2 和 8 分别对应 V 分量第一部分的源和目标，3 和 9 则分别对应 Y 分量第二部分的源和目标（U、V 没有第二部分）。Start_of_active_Video_Frame[X]给出了各活动视频帧的起始地址。

完成上述初始化和设置后，对每一帧视频数据，编码过程如下。

（1）等待接收到一个填满的 4-2-0 缓冲区，其中包含单独的 Y、Cr 和 Cb 分量。

（2）将缓冲区指针传递给 MJPEG 编码器：

```
JPEG_McuBuffer_CONFIG( mcu_ex, MCU_POINTER_C1,
                        ( unsigned int)lInputBuffer);
```

（3）执行 MJPEG 编码过程：

```
JPEG_EncodeSequentialImage( lJpegEnc, & NumBytes );
```

（4）向 AVI 文件写入数据流：

```
MJPEG_AVI_WriteStream(lStreamHandle, StreamBuffer, NumBytes, 0);
```

也可通过 USB 等接口传到主机 PC 进行保存。

（5）回到第（1）步，等待编码下一帧。

完成编码后，需要关闭 AVI 流和输出文件（如果有），释放被销毁使用的资源：

```
JPEG_Encoder_DELETE(lJpegEnc);
JPEG_MemAlloc_DELETE(StreamBuffer_Obj);
```

如果要连续编码多帧，必须要采用乒乓缓冲技术，否则正在被编码的帧会同时被写入数据，这样就会导致数据被破坏。采用乒乓缓冲技术就能够做到正在编码的前一帧不会被下一帧数据破坏，如图 4.28 所示。

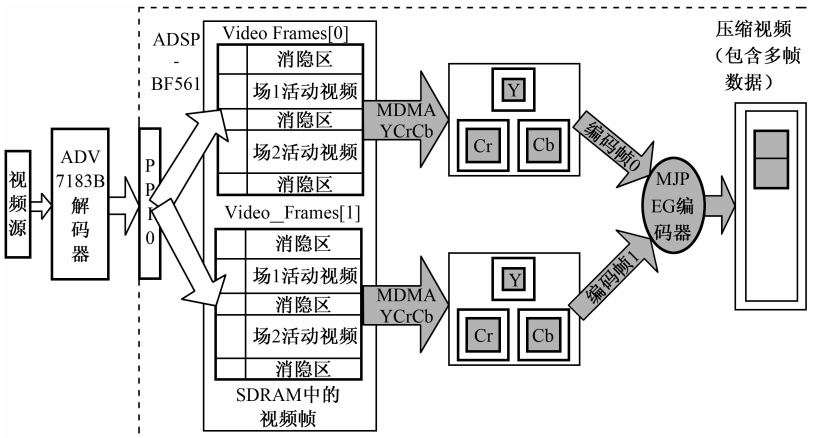


图 4.28　在 MJPEG 编码中使用乒乓缓冲

为了将数据永久保存，可以通过 USB 接口将压缩后的数据传输到 PC 上进行保存。利用 ADI 提供的 USB EZ LAN 控制板，可以实现 Blackfin 到主机的基于 USB2.0 的简单数据传输。ADI 提供对 USB EZ LAN 控制板的完整的硬件和软件支持，可用于 ADSPBF561 等 EZ-Kit。

通常，主机 PC 和 Blackfin 处理器之间会相互发送命令块。这里，Blackfin 将接收 PC 的 USB 命令块并执行相应的动作，如图 4.29 所示。相关的命令和参数定义在文件 USB Command.h 中。

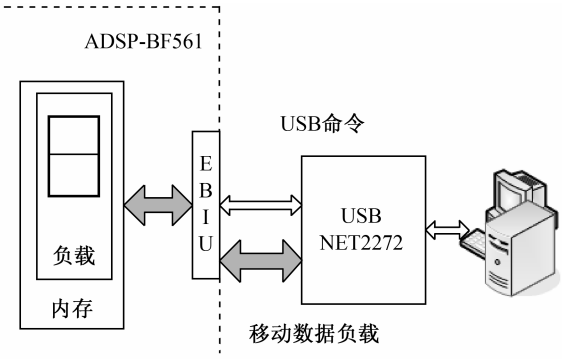


图 4.29　主机与 Blackfin 处理器间的 USB 传输

将上述所有模块组合起来，就构成了一个完整的包括视频采集与回放、MJPEG 编码、USB 传输的视频编码与回放系统，如图 4.30 所示。硬件上分为 4 大部分。首先是视频输入组件，主要是 ADV7183B；其次是视频输出组件，使用了 ADV7179；第三部分是 MJPEG 编码，它接收视频帧数据，利用内存 DMA 进行分量分离，再编码为压缩视频数据；最后一部分是通过 USB 与主机的传输，将压缩后的数据保存到 PC 文件中。PC 还可通过 USB 传

输命令控制 Blackfin 处理器的动作。

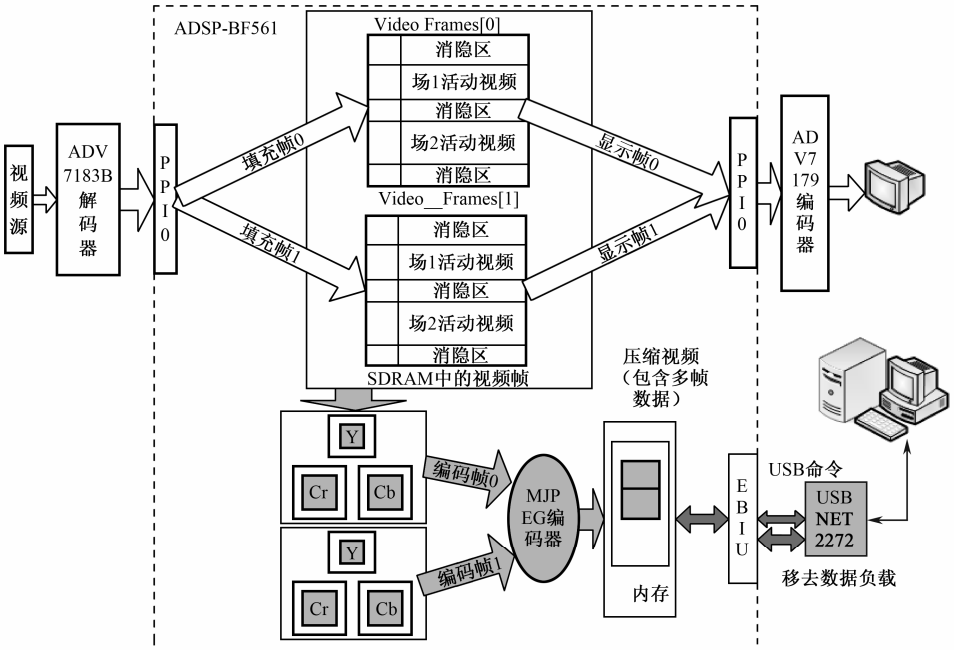


图 4.30　完整的视频采集回放与编码系统

软件流程图如图 4.31 所示。本软件流程清晰明了，应用程序首先建立系统服务，然后初始化设备驱动器。Blackfin 处理器软件接收主机通过 USB 发送过来的命令块，根据命令的内容执行对应的函数。如果是执行 MJPEG 编码命令，Blackfin 处理器就开始编码，直到 PC 通知它停止。

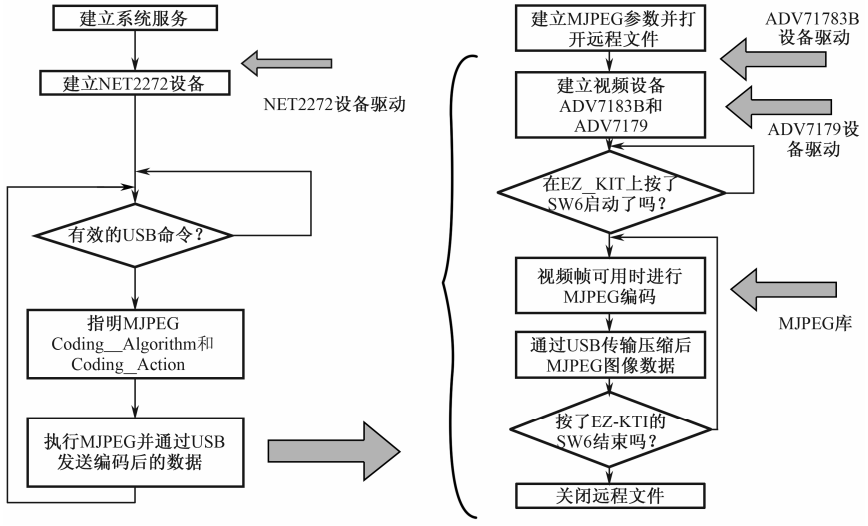


图 4.31　视频采集回放与编码系统流程图

建立视频驱动器的代码参见函数 Setup_Video_Driver()，在其中打开了视频编解码器 AVD7179 和 DAV7183，并分别分配了乒乓缓冲区 Out1_Buffer2D、Out2_Buffer2D、In1_Buffer2D 和 In2_Buffer2D，指定了回调函数 PPI_Callback_Out()和 PPI_Callback_In ()，最后启动了输入和输出数据流。

PPI_Callback_In()调用用于定义的 ISR 函数 PPI_In_User_Callback()，在接收到事件 ADI_DEV_EVENT_BUFFER_PROCESSED 时，根据参数 pArg 的值确定刚刚完成的帧的序号，配置 MDMA 描述子链中的起始地址，并调用 adi_dma_Queue()向 DMA 通道添加描述子。DMA 传输结束后会调用中断服务函数 MDMA1_Callback()，进而调用用户定义的 MDMA1_User_Callback()。

如果对应的触发事件是 ADI_DMA_EVENT_DESCRIPTOR_PROCESSED，则通过增加 YUVBufferReadIndex 的值。MJPEG 编码器（函数 MJPEG_encode 中）正是等到该信息后才开始新一帧数据的编码的。

PPI_Callback_Out()中调用用户定义的 PPI_Out_User_Callback()函数，它在接收到 ADI_DEV_EVENT_BUFFER_PROCESSED 事件后，根据 pArg 参数的值确定上一完成帧序号，配置输出 DMA 描述子链中的起始地址，并调用 adi_dma_Queue()向输出 DMA 通道添加描述子，启动数据流输出，完成视频显示任务。

再来进一步看编码过程。接收到编码命令后，建立 MJPEG 参数，打开主机 PC 上的远程文件。然后建立视频编解码器所需的设备驱动器。启动视频流后，Blackfin 处理器就会接收到视频信息，该视频信息同时会传输到本地监视器进行显示，然后通过开关或按钮启动视频编码。此时 Blackfin 处理器就开始对视频帧进行编码，并通过 USB 将压缩结果传输到主机 PC 上，直到用户通知它结束。

在此对硬件做一个小结。首先需要 BF561 EZ-Kit 及 USB EZ LAN 扩展板卡、运行 Windows XP 的主机 PC（需要 USB 2.0 端口）、一个视频源（可以使用 DVD 播放器），还有视频显示器件。此外还需要安装 VisualDSP++，使我们能够使用设备驱动器和系统服务。我们要将 EZ LAN 扩展板连接到 EZ-Kit，然后通过 USB 2.0 端口将 PC 连接到扩展板。同时要注意，正确地设置 EZ LAN 扩展板和 EZ-Kit 上的开关，确保 USB 和 Blackfin 按照预期进行通信。其连接示意图如图 4.32 所示。

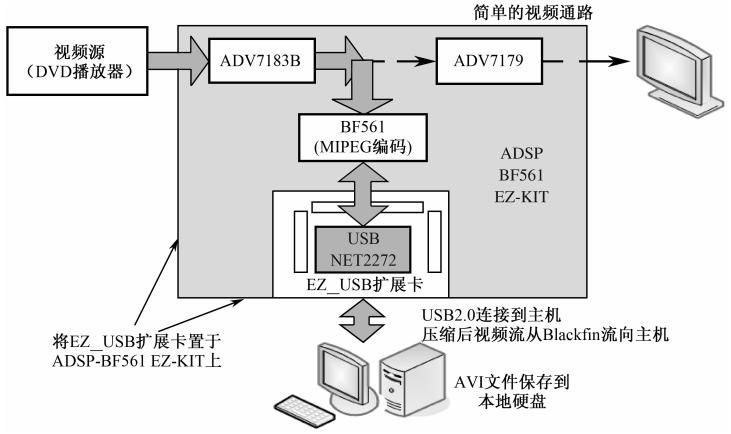


图 4.32　视频采集回放编码系统硬件连接示意图

4.9 视频 Sobel 边缘提取系统

下面介绍视频 Sobel 提取系统的实现，展示更多的算法实现细节。它从模拟视频流采集视频数据，然后提取每帧图像的 Sobel 边缘，并将边缘提取结果显示到视频监视器上。外设主要包括 ADV7183 视频解码器、ADV7171 视频编码器。控制功能主要指软开关（SW），通过按下并释放某个 SW 可以触发某个动作，如开始或停止边缘提取、显示等操作。

系统的实现主要来自 Blackfin SDK 的支持。在前面介绍视频输入、输出基本知识的基础上，本实现过程主要解决了如下几个问题。

（1）为视频输入、视频输出准备不同的缓冲区，因为二者的内容不再是相同的，其中输出视频是边缘提取的结果。

（2）视频输入和视频输出数据都是存储在外部（L3）内存中的，为了提高处理效率，Sobel 算法的实现应当尽量利用 L1 内存。为此在 L1 内存中为 Sobel 算法的输入和输出分别申请一段内存，其尺寸远小于视频的原始尺寸。

（3）在处理过程中，需要首先将视频输入缓冲区的数据分批次（分成多个矩形块）传输到 L1 中的 Sobel 输入缓冲区中（DMA 方式），处理后存放到 Sobel 输出缓冲区，紧接着传输到输出缓冲区适当位置。

（4）待所有矩形块数据处理完毕并且传输到视频输出缓冲区后，即可对其进行显示了。

为此，我们首先定义如下一些重要的常量。

```
// 每一行中的有效视频数据分为 6 段（NTSC 和 PAL）——水平方向
#define  SOBEL_ACTIVE_VIDEO_BLOCKS       6
// NTSC 视频帧有效场分为 3 段—垂直方向
#define  SOBEL_FIELD_BLOCKS_NTSC         3
// PAL 视频帧有效场分为 4 段—垂直方向
#define  SOBEL_FIELD_BLOCKS_PAL          4
// 每个有效场包含的 Sobel 子块的数目（NTSC）
#define  SOBEL_BLOCK_COUNT_NTSC
         (SOBEL_ACTIVE_VIDEO_BLOCKS * SOBEL_FIELD_BLOCKS_NTSC)
// 每个有效场包含的 Sobel 子块的数目（PAL）
#define  SOBEL_BLOCK_COUNT_PAL
         (SOBEL_ACTIVE_VIDEO_BLOCKS * SOBEL_FIELD_BLOCKS_PAL)
// Sobel 算法的偏移量(sobel.h)
// Sobel 输出缓冲区中第一行和最后一行都是无效的
// Sobel 输出缓冲区中第一列和最后一列都是无效的
#define  SOBEL_OFFSET                    2
// Sobel 缓冲区列的数目（仅对有效数据）
#define  SOBEL_COLUMN_SIZE
(ITU_PIXEL_PER_LINE / SOBEL_ACTIVE_VIDEO_BLOCKS)
```

```
// Sobel 缓冲区的尺寸（给出最大值而不用动态申请方式）
#define  SOBEL_BUF_SIZE  ((SOBEL_COLUMN_SIZE+SOBEL_OFFSET) \
*((NTSC_ACTIVE_FLINES/SOBEL_FIELD_BLOCKS_NTSC)+SOBEL_OFFSET))
// ************************************************************
ITU 656 视频帧上边缘检测相关定义
************************************************************/
// 视频输入、视频输出帧缓冲区尺寸——设为最大值
#define  VIDEO_BUF_SIZE              PAL_VIDEO_FRAME_SIZE
```

此外还需要定义以下一些相关的变量

```
/************************************************************
视频数据缓冲区
************************************************************/
// 定义视频数据处理所需缓冲区
// 因为 SDRAM 的性能原因，每个缓冲区必须位于不同的 bank 上
section("video_sdram_bank0") volatile u8 Video_In_Buf0 [VIDEO_BUF_SIZE];
section("video_sdram_bank2") volatile u8 Video_Out_Buf [VIDEO_BUF_SIZE];
// Sobel 边缘检测缓冲区
section("Sobel_In_buf0") volatile u8 Sobel_In_Buf0      [SOBEL_BUF_SIZE];
section("Sobel_In_buf1") volatile u8 Sobel_In_Buf1      [SOBEL_BUF_SIZE];
section("Sobel_Out_buf0") volatile u8 Sobel_Out_Buf0  [SOBEL_BUF_SIZE];
section("Sobel_Out_buf1") volatile u8 Sobel_Out_Buf1  [SOBEL_BUF_SIZE];
// 视频输入缓冲区属性
static   ADI_DEV_2D_BUFFER Video_In_Buffer2D [VIDEO_BUFFERS];
// 视频输出缓冲区属性
static   ADI_DEV_2D_BUFFER Video_Out_Buffer2D[VIDEO_BUFFERS];
/**********************************
MDMA 句柄及数据
**********************************/
// DMA 流句柄
// MDMA 流——从视频输出缓冲区到 Sobel 输入缓冲区
ADI_DMA_STREAM_HANDLE        MDMA_Handle_Stream0;
// MDMA 流——从 Sobel 输出缓冲区到视频输出缓冲区
ADI_DMA_STREAM_HANDLE        MDMA_Handle_Stream1;
// MDMA 2D 传输数据类型
// 定义视频输出缓冲区到 Sobel 输出缓冲区的数据结构
ADI_DMA_2D_TRANSFER          MDMA_Sobel_In_Src;
ADI_DMA_2D_TRANSFER          MDMA_Sobel_In_Des;
// 定义 Sobel 输出缓冲区到视频输出缓冲区的数据结构
```

```
ADI_DMA_2D_TRANSFER          MDMA_Sobel_Out_Src;
ADI_DMA_2D_TRANSFER          MDMA_Sobel_Out_Des;
```

下面介绍算法的主要实现步骤。

在主程序 main()中，首先要完成一系列的初始化工作，然后循环地在接收到视频输入后执行边缘提取与显示工作。

初始化工作包括以下几个部分。

（1）初始化系统：主要是初始化 EZ KIT 开发板，使能视频解码器 ADV7183。

（2）安装系统服务：包括初始化中断管理器，初始化按钮状态等。

（3）安装视频解码器：打开视频解码器 ADV7183 并对其进行配置，设置视频输出帧格式，为 Sobel 边缘检测配置并安装 MDMA 等。

（4）启动视频解码器 ADV7183 的数据流，开始捕获视频数据。

下面的代码实现了 MDMA 的安装与配置。

```
section ("App_Code_L1") // 生成的代码将放到 App_Code_L1 段
void ConfigureMDMA() {
   // 配置 MDMA 2D 结构
   /*****************************************
   视频输入缓冲区到 Sobel 输入缓冲区流
   *****************************************/
 /**** Sobel 输入流目的缓冲区配置 ****/
 MDMA_Sobel_In_Des.XCount    = (SOBEL_COLUMN_SIZE +
                               SOBEL_OFFSET);
 // 只复制亮度值
 MDMA_Sobel_In_Des.Xmodify = 1; // 1 字节递增值
 MDMA_Sobel_In_Des.Ycount = (SobelRowSize+SOBEL_OFFSET);
 // 移到下一行
 MDMA_Sobel_In_Des.Ymodify = 1;
/**** Sobel 输入流源缓冲区配置 ****/
 MDMA_Sobel_In_Src.Xcount = (SOBEL_COLUMN_SIZE
                            + SOBEL_OFFSET);
 // 只复制亮度值
 MDMA_Sobel_In_Src.Xmodify = 2; // 2 字节递增值（二选一）
 MDMA_Sobel_In_Src.Ycount = (SobelRowSize+SOBEL_OFFSET);
 // 移动到下一行，指向第一个亮度位置－2
 MDMA_Sobel_In_Src.Ymodify = (DataPerLine -
((SOBEL_COLUMN_SIZE + SOBEL_OFFSET) * 2) + 2);
   /*****************************************
   Sobel 输出缓冲区到视频输出流
   *****************************************/
   /**** Sobel 输出流目的缓冲区配置 ****/
```

```
 // 只复制有效数据
 MDMA_Sobel_Out_Des.Xcount = SOBEL_COLUMN_SIZE;
 // 只复制有效亮度值
 MDMA_Sobel_Out_Des.Xmodify = 2;
 // 只复制有效数据
 MDMA_Sobel_Out_Des.YCount          = SobelRowSize;
 // 移动到下一行，指向第一个亮度位置处
 MDMA_Sobel_Out_Des.Ymodify = (DataPerLine -
                               (SOBEL_COLUMN_SIZE * 2) + 2);
   /**** Sobel 输出流源缓冲区配置 *****/
 MDMA_Sobel_Out_Src.XCount          = SOBEL_COLUMN_SIZE;
 MDMA_Sobel_Out_Src.XModify         = 1;
 MDMA_Sobel_Out_Src.YCount          = SobelRowSize;
 // 跳过当前行的最后一列和下一个 row + xmodify 行的第一列
MDMA_Sobel_Out_Src.Ymodify = 3;
 /*********************************************************
   安装 MDMA 视频输出缓冲区到 Sobel 输入缓冲区流
   *********************************************************/
   // 打开 MDMA
     // DMA 管理器句柄，控制器—MDMA 流 0，客户句柄，流句柄，回调函数
     ezErrorCheck ( adi_dma_MemoryOpen ( DMAManagerHandle,
        ADI_DMA_MDMA_0, NULL, &MDMA_Handle_Stream0, NULL) );
   /*********************************************************
   安装 MDMA 的 Sobel 输出缓冲区到视频输出缓冲区流
   *********************************************************/
     // 打开 MDMA
     // DMA 管理器句柄，控制器—MDMA 流 1，客户句柄，流句柄，回调函数
     ezErrorCheck( adi_dma_MemoryOpen ( DMAManagerHandle,
ADI_DMA_MDMA_1, NULL, &MDMA_Handle_Stream1, NULL));
}
```

接下来要初始化视频输入到 Sobel 输入流。

```
section  ("Callback_Code_L1")
void SobelInInit (u32 *pBuffer)
{
 // 初始化视频输入/输出缓冲区地址指针
 if (pBuffer == (u32 *) (&Video_In_Buffer2D[0]))
 { // 意味着 Video_In_Buf0 已准备好做 Sobel 变换
  // 起始地址：指向 Video_In_Buf0 活动场 1 第一个亮度值—2
```

```
        pVideoInBuf =(u8*)(&Video_In_Buf0[0] + ( Field1Skip +
    ActiveVideoSkip - (DataPerLine + ITU_Y_OFFSET)));
     }
     else
         return;          // 该缓冲区不是期望值
     // 视频输出起始地址：指向 Video_Out_Buf 活动场 1 第一个亮度值
     pVideoOutBuf = (u8*)(&Video_Out_Buf[0] + (Field1Skip +
    ActiveVideoSkip + ITU_Y_OFFSET));
     tpVideoInBuf    = NULL; // 清除临时视频输入缓冲区指针
     tpVideoOutBuf   = NULL; //清除临时视频输出缓冲区指针
     // 初始化 Sobel 缓冲区管理标记和计数器
       SobelFlag.SobelInBufID = 1; // 开始将亮度值载入 Sobel_In_Buf0
       SobelFlag.SobelOutBufID = 1;      // 将处理过的亮度值载入 Sobel_Out_Buf0
     SobelFlag.SobelInBuf0Done = TRUE;  // Sobel_In_Buf0 可接收亮度值块了
     SobelFlag.SobelInMDMALock = FALSE; // 解锁视频输入到 Sobel
                                        // 输入 MDMA 流
     SobelFlag.SobelOutBuf0Done = FALSE;// 标记两个 Sobel 输出缓冲区
                                        // 已准备好接收边缘检测数据
     SobelFlag.SobelOutBuf1Done = FALSE;
     SobelFlag.SobelOutMDMALock = TRUE; // 锁住 Sobel 输出到视频输出
    // MDMA 流      SobelFlag.InitSobelOut = FALSE; // Sobel 输出函数未初始化
     // ……
    }
```

Sobel 输出到视频输出流的操作与上面过程类似。为了进行 Sobel 变换，需要首先将视频输入缓冲区（L3）部分内容传输到 Sobel 输入缓冲区（L1）以提高处理速度。下面这个函数在回调函数中被调用。

```
section  ("App_Code_L1")
void MDMA_SobelIn(void)
{
 /**** Sobel 输入流目的缓冲区配置 ****/
 // 检查上一个使用的 Sobel 输入缓冲区 ID
 // 如果上一个缓冲区是 Sobel_In_Buf1，则接下来要要使用 Sobel_In_Buf0
 if (SobelFlag.SobelInBufID) {
     if (SobelFlag.SobelInBuf0Done) { // 检查 Sobel_In_Buf0 是否就绪
         MDMA_Sobel_In_Des.StartAddress =
                                          (void *) (&Sobel_In_Buf0[0]);
         // Sobel_In_Buf0 将被填充上亮度值
         SobelFlag.SobelInBufID = 0;
     }
```

```
        else     return; // 返回，Sobel_In_Buf0 尚未就绪
    }
    else { // 前一个缓冲区是 Sobel_In_Buf0，那么下一个是 Sobel_In_Buf1
        if (SobelFlag.SobelInBuf1Done)
        {
            MDMA_Sobel_In_Des.StartAddress =
                                        (void *) (&Sobel_In_Buf1[0]);
            SobelFlag.SobelInBufID = 1;
        }
        else return;
        }
     ++SobelInBlockCount; // 下一个 Sobel 块的号码
     // 检查这是否是第一个 Sobel 块，如果是，载入 pVideoInBuf 值
       if (tpVideoInBuf == NULL){
         tpVideoInBuf = pVideoInBuf;
         SobelInBlockCount = 0; // Sobel 正在处理块 0
       }
     // 检查活动场 1 是否已结束。如果是则处理 Video_In_Bufx 活动场 2
       else if (!(SobelInBlockCount % SobelBlockCount)) {
           pVideoInBuf = (pVideoInBuf + (Field2Skip - Field1Skip));
           tpVideoInBuf = pVideoInBuf;
           SobelInColumnBlockCount = 1;  // 移动到 Sobel 块的第一列
       }
       else if (!(SobelInBlockCount % SobelFieldBlocks)) { // 移到下一个列块
           tpVideoInBuf = (pVideoInBuf
    + (SOBEL_COLUMN_SIZE*2* SobelInColumnBlockCount));
           SobelInColumnBlockCount++; // 移到 Sobel 块的下一列
       }
       else // 移到 Sobel 块的下一行
           tpVideoInBuf = (tpVideoInBuf + (DataPerLine * SobelRowSize));
     MDMA_Sobel_In_Src.StartAddress = (void *) tpVideoInBuf;
     // 锁住视频输出到 Sobel 输入的 MDMA 直到这个传输结束
     SobelFlag.SobelInMDMALock       = TRUE;
         // 启动 MDMA 2D 复制（视频输入缓冲区到 Sobel 输入缓冲区流）
         ezErrorCheck( adi_dma_MemoryCopy2D ( MDMA_Handle_Stream0,
    &MDMA_Sobel_In_Des,
    &MDMA_Sobel_In_Src,1,MDMA_Callback0));
    }
```

完成一个 Sobel 块的边缘提取后，将处理结果由 Sobel 输出缓冲区传输到视频输出缓冲

区是一个相反的过程，其处理过程刚好与上面相反。

为了充分利用 Blackfin DSP 处理器的性能，需要采用汇编语言实现 Sobel 边缘快速计算。所采用的水平和垂直掩码如图 4.33 所示。

−1	−2	−2
0	0	0
1	2	1

水平方向掩码

−1	0	1
−2	0	2
−1	0	1

垂直方向掩码

图 4.33　Sobel 算子掩码

函数原型为 void _sobel_fast(unsigned char* in, int row, int col, unsigned char *out, int threshold)。其中 in 为输入图像的指针，row 和 col 为输入图像的行数和列数，out 为输出缓冲区指针，threshold 为设定的阈值。以下算法中用到的寄存器包括 A0、A1、R0～R7、I1、I3、B0～B3、M0～M3,、L1、P0～P5、LC0 和 LC1。

```
.section     L1_code;
.global      _sobel_fast;
.align       8;
_sobel_fast:
    [--SP]=(R7:4,P5:3);       // 将 R7:5 和 P5:4 入栈(Push)
    L1 = 0;
    P5=R0;                    // 输入图像地址
    P0=R1;                    // 行数
    P1=R2;                    // 列数
    R3=R2≪1 ||P4=[SP+40];     // 得到输出地址
    M0=R2;                    // m0=列数
    nop;     nop;     nop;     nop;
    P0+=-2;                   // 行数-2
    M3=P0;
    R3+=2;                    // 2*列数+2
    R4=R1.L*R2.L(is)||R6=[SP+44]; // 得到阈值
    P2=R3;                    // P2=2*列数+2
    R7=1; R0=-1; R1=-2;
    R2=2;                     // 初始化 R2 等于 2 以在栈上保存参数
    SP+=-16;                  // 减少栈指针保存参数
    I1=SP; B1=SP;             // 将 I1 和 B1 设为 SP
    B2=R4;                    // 行数*列数
    R4=R4≪2||W[I1++]=R0.L;    // 4*行数*列数
    B3=R4;    P3=R4;
    W[I1++]=R1.L;             // 将水平和垂直方向的掩码存到栈中
    W[I1++]=R7.L;
```

```
    W[I1++]=R1.L;
    W[I1++]=R2.L;
    W[I1++]=R7.L;              // 保存所有参数到栈上
    R6=R6.L*R6.L(IS)||W[I1++]=R0.L;  // 得到阈值的平方
    R5=R7-R7(NS)|| W[I1++]=R2.L;     // R5 清零
    I1=B1; L1=16;                    // 将 L1 设为 16 实现循环缓冲区
    R0=B[P5++] (Z) | | R1.L=W[I1++]; // 得到第一个输入和参数
    I3=P4;
    P3=B2;                     // 行数*列数
    B0=SP;
    MNOP;
    R4 = 254;                  // 边缘像素
    LSETUP(FIRST_ROW,FIRST_ROW) LC0=P1;  // 循环计数器 == 列数
FIRST_ROW:
    B[P4++]=R5;
    P1+=-2;                                // 列数-2
    LSETUP(ROW_ST,ROW_END) LC0=P0;         // 循环计数器 == 行数-2
    P0=SP;
    P3=B0;
ROW_ST:
       B[P4++]=R5;
       LSETUP(COL_ST,COL_END)LC1=P1;       // 循环计数器=列数-2
COL_ST:
      A1=R0.L*R1.L, A0=R0.L *R1.L(IS)  || R0=B[P5++](Z)||R1.H=W[I1++];
// A1=-x00,A0=-x00,取 x01 和-2
A1+=R0.L*R1.H(IS) ||R0=B[P5](Z)||R1.H=W[I1++];
P5=P5+P1;                                  // 指向下一行起始位置
A1+=R0.L*R1.L ,A0+=R0.L *R1.H(IS) || R0=B[P5++](Z)||R1.L=W[I1++];
A0+=R0.L*R1.L(IS)||R0=B[P5++](Z)|| R1.H=W[I1++];
R5=R1-R1(NS)||R0=B[P5](Z)||R1.L=W[I1++];
P5=P5+P1;
A0+=R0.L *R1.H(IS)||R0=B[P5++](Z)||R1.H=W[I1++];
A1+=R0.L*R1.L,A0+=R0.L*R1.H(IS)|| R0=B[P5++](Z)||R1.H=W[I1++];
A1+=R0.L *R1.H(IS)||R0=B[P5++](Z);
R3=(A1+=R0.L*R1.L),R2=(A0+=R0.L*R1.L)(IS);
R3=ABS R3;       R2=ABS R2;
P5-=P2; // 指针移回原处，准备处理下一组数据
R3=R3+R2(ns)||R0=B[P5++](z)||R1.L=W[I1++]; // R3 包含最终比较结果
CC=R3<=R6;                 // 结果是否比阈值小
```

```
R3=R3≫2;
IF !CC R5 = R3;
COL_END:    B[P4++]=R5;        // 保存结果
       P5+=1;                  // 移动到下一行的起始位置
       R5=R1-R1(ns)||R0=B[P5++](Z);
ROW_END:    B[P4++]=R5;        // 最后的元素存为 0
P0=M0;                         // p0==列数
LSETUP(END_ROW,END_ROW) LC0=P0; // 清除最后一行
END_ROW:
nop; B[P4++]=R5;
P3=B3; P1=B2;
L1=0;                          // 清除 L0 以避免循环缓冲区问题
   SP+=16;
   P3=P3+(P1≪1);               // 偏移值使栈指针移到正常位置
   (R7:4,P5:3)=[SP++];         // 参数出栈
   RTS;
_sobel_fast.end:
```

到此为止，我们通过这个例子较为详细地介绍了基于 Blackfin 处理器的视频处理系统的开发过程，列出了一些面临的主要问题和典型解决策略。在此基础上，后面几章具体讨论几种常用的视频处理技术及应用。

为了能够尝试这些例子，除了准备好相关的硬件外，还要安装 VisualDSP++，另外还要下载并安装 Blackfin SDK [40]，其中包含了更多的多媒体处理例程。更多信息请进一步参考 ADI 公司的官方网站 www.analog.com。

第5章　视频应用设计原则及基础应用简介

Blackfin 处理器为高性能应用提供了多种高效的数据管理方法，以及基于不同算法的子处理块粒度。本章描述了一组典型模板，可以用来在 Blackfin 处理器上有效地管理多媒体数据。这些模板可以作为用户开发特定应用程序的起点，它们可以被修改以适应特定应用的需求。

在多媒体应用中，内存管理和传输设计是影响系统性能的两个至关重要的因素。对这两个方面的相关内容，本章讨论了其一般原则和常见的处理方法。

最后简单介绍了一些基础性的、简单的视频处理应用。

5.1　视频应用开发模板

多媒体应用中所需的缓冲区尺寸往往超过处理器的内部内存空间。为了充分利用处理器片上内存（L1/L2）的低时延优点，需要考虑将外设数据直接移动到 L1 或 L2 内存来进行处理，这样的处理模式就形成了几种常见的开发模板[47]。这时还要考虑应用程序中图像块所需内存尺寸和处理粒度之间的折中。本质上，这些模板都利用了多媒体应用所固有的可预测数据访问模式，能够最小化在不同级别内存之间的数据传输量，因而提高资源利用率。

5.1.1　视频开发模板综述

视频序列可以看做是一个三维信号，包含两个空间维度（即单个二维帧的维度）和一个时间维度（帧序列）。因而，视频数据流可以按以下两种维度划分：一个是时间维度，一个是空间维度。在时间维度上，数据流可以按照“图片组”（GOP）或帧的粒度进行管理。在空间维度上，数据移动可以在扫描线或宏块级别上进行管理。视频信号的这些不同级别的粒度成为子处理块。显然，上述各种子处理块的粒度从 GOP 到行级别是逐渐递减的。

这些模板利用视频信号固有的可预测数据访问模式的优点，可以隐藏内存的延迟，降低视频数据在不同级别内存之间的移动。在大多数图像处理算法中，子处理块的数据访问模式是可预测的。例如，在 JPEG 压缩中，图像中的宏块逐行地顺序被访问。表 5.1 显示了一些其子处理器块访问模式可预测的例子。

表 5.1　可预测的访问模式

应用程序示例	注　　释
颜色转换（如 YUV 到 RGB）	图像逐行被访问
直方图分析	图像数据逐行被访问
边缘检测	宏块逐行地顺序被访问
JPEG/运动 JPEG	宏块逐行地顺序被访问
运动检测	宏块逐行地顺序被访问，独立于上一帧
MPEG2/MPEG4	运动窗口顺序地通过宏块，独立于前后帧

如果帧图像的数据访问模式是事先已知的，数据可以在被内核请求前移动到 L1 或 L2 内存。这就避免了内核因为内存访问被拖延导致的额外时延。而且，为了隐藏内存传输的延迟，可以有效地使用 DMA，这样就可能在“背景”中完成数据传输。使用 DMA 而非 Cache 可以节省由于 Cache 未命中而导致的内核时钟消耗，这对实时性系统至关重要。

为了进一步节省系统资源，模板通过将图像数据从外设移动到片上内存来使用更小、更快的 L1/L2 内存空间，这样能够降低可能由访问外部 SDRAM 导致的延迟。这一点适用于那些具有较小子处理块粒度的算法（如扫描线或宏块）。

设计模板的目标是，一旦应用程序的子处理块定义好了，而且满足了实时限制，图像处理算法就会落入其中某个模板，这样就可以快速得到有效的设计布局参考。每个模板都有具体的例子来展示。在某些算法中，在空域的子处理块之间可能存在设计依赖（如当前参考帧与前一帧或后一帧之间）。在这些情况下，管理数据就更加复杂了。

5.1.2　视频开发模板类型

下面简要描述基于不同子处理块粒度的 3 个不同的模板，包括空间域子处理块（行和宏块）模板，以及时域上存在相关性的应用程序模板。

1．扫描线处理

对于扫描线处理，一种方法是从外设中一次性将一帧数据采集到外设中。采集到一帧数据后，就可以执行逐行地将数据传输到 L1 内存，可以使用 Cache 或 DMA。一种更有效的方法是将一行数据直接从外设读入 L1 数据内存，接下来将处理过的该行数据直接从 L1 内存传输到外设或外部内存。这就降低了所需的内存传输量，并节省了所需的外部总线带宽。

图 5.1 显示了基于扫描线模板中的数据流。DMA 用来将图像数据直接从外设传输到 L1 内存。图像处理后通过 DMA 送到外设。L1 内存中应采用乒乓缓冲，以允许 DMA 和内核的同时访问，当然还需要避免 L1 子块的竞争。

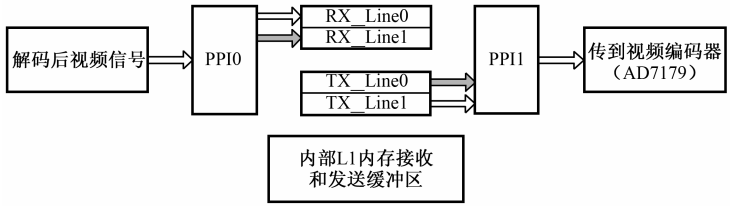


图 5.1　扫描线处理模板数据流

2．宏块处理模板

对于宏块处理，假设其高度和宽度分布是 *n* 和 *m*。L1 数据内存不足以一次性存储图像的 *n* 行完整数据，所以必须使用 L2 或外部内存。为了节省外部总线带宽，最好尽量使用 L2 内存。

因为 L2 内存无法放下整幅图像的数据，所以一次只能将 *n* 行图像数据从外设（PPI）传输到 L2 内存。然后宏块处理再由 L2 内存传输到 L1 数据内存。在 L1 和 L2 内存中都采用乒乓缓冲，数据传输采用 DMA 通道。图 5.2 显示了该模板的数据流。同时，在该模板中，外部内存可以用来作为输入或输出帧缓冲，以确保对资源更一致的使用。图 5.2 中的 L2 Tx 或 L2 Rx 缓冲区可以放到外部内存中。

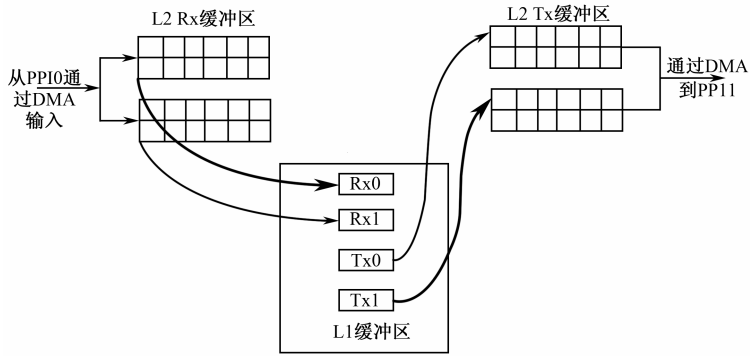


图 5.2　宏块处理模板数据流示意图

3．帧间处理模板

该模板用于子处理块间在时间维度上存在依赖关系的应用，如在前后帧之间。L1 和 L2 内存空间不足以存放相互依赖的帧数据。这样，外部 SDRAM 内存用来映射依赖帧。依赖帧的子处理块从外部内存传输到 L1 内存。如果 L1 内存不够用，可以使用 L2 内存存放中间缓冲区。

图 5.3 显示了一个与前一帧相关的基于宏块处理的应用程序。其中，当前帧和参考帧指针在每来一个新帧时要进行一次交换，即使前一个“当前帧”成为下一个“参考帧”。对于基于扫描线的前后帧相关的应用程序，可以在 L1 和外部内存之间传输行数据。另外，依赖性也可以存在于几个子处理块之间。这种情况下，帧的一片数据（一组扫描线数据）可

以传输到内部内存。

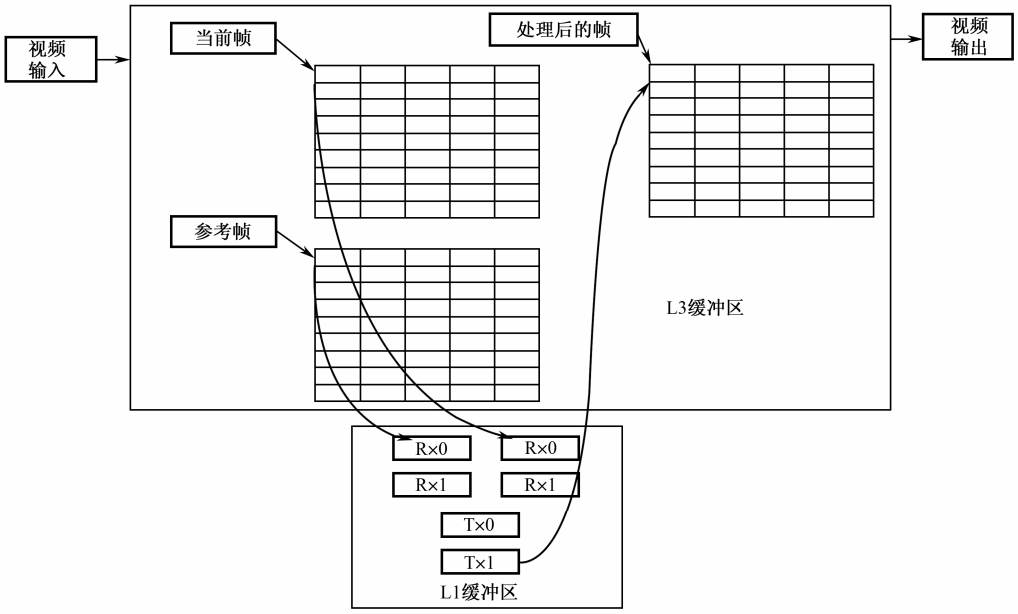


图 5.3　帧间处理器模板数据流示意图

5.1.3　针对 Blackfin 处理器的优化

为了进一步提高性能和系统带宽使用的效率，这些模板中采用了几种优化技术。主要的优化方法如下：

（1）采用 16/32 位传输。所有外设 DMA 和 MemDMA 传输都采用最大总线带宽。ADSP-BF53x 处理器采用 16 位数据总线，ADSP-BF56x 处理器采用 32 位数据总线。

（2）有效地使用 DMA 通道。在同一个 DMA 通道上不要同时启动两个内存传输。

（3）SDRAM 子块划分。SDRAM 划分为 4 个子块，以确保同时访问多个帧缓冲区并具有最小的方向切换时间。例如，ADSP-BF561 EZ-KIT Lite 板有 64M 字节的 SDRAM，可以配置为 4 个 16M 字节的内部 SDRAM 子块。为利用子块结构的优势，需要同时访问的缓冲区不要映射到同一个 SDRAM 子块上。

（4）DMA 流量控制。DMA 流量控制寄存器可以用于有效地使用系统带宽。

5.1.4　使用视频开发模板

为了针对具体应用使用模板，首先要确认以下内容：

（1）图像处理算法中子处理块的粒度，用来选择具体的模板。

（2）可用的 L1 和 L2 数据内存，以及具体模板所需要的片上内存

（3）估计每子处理块所需时钟周期。这可以帮助确定使用该开发模板能否满足应用程

序的实时性限制。

（4）子处理块间的空间和时间依赖性。如果存在依赖性，需要修改模板以适应数据依赖性。

如表 5.2 所示为每个模板的子处理块所需的内核时钟周期。表中列出了 Debug 和 Release 模式下的时钟周期。对于帧间处理，在依赖帧之间需要多次传输，与扫描线或宏块处理相比，就减少了可用的内核时钟周期。注意，帧间处理所用的时钟周期只是对于传输一段帧数据（一组扫描线数据）到内部内存。

表 5.2　每个模板的详细说明

模　　板	CCLK/像素（Debug）	CCLK/像素（Release）	所需 L1 数据内存	所需 L2 数据内存	注　　释
行处理	36	42	2×行尺寸	N/A	采用循环连接缓冲区；L1 中采用双缓冲区
宏块处理	30	36	2×宏块尺寸 宏块尺寸=n×m	宏块高度×2×行尺寸 （帧片段的尺寸）	采用循环连接缓冲区；L1 和 L2 采用双缓冲区
帧间处理	30	35	子处理块尺寸×依赖块数目	子处理块尺寸×依赖块数目	只能用 L1 或 L2； 采用循环连接缓冲区；L1 或 L2 采用双缓冲区

因为大多数数据访问是在 L1 内存中执行的，因此由内存延迟导致的时钟周期数做到了最小化。因此这里显示的时钟周期数完全可以作为算法所需内核计算量的预算量。算法对应的内核处理所需时钟周期，或者根据理论进行计算，或者利用 Blackfin 处理器上的时钟周期计数器来获得。VisualDSP++中的模拟或仿真环境可以用来获取该信息。

表 5.2 还显示了所需的 L1 和 L2 数据内存尺寸，模板中采用缓冲区循环链接策略。对于缓冲区循环链接的更多知识，参考 VisualDSP++中的 DMA 驱动器和系统服务手册。

5.1.5　视频开发模板应用举例

每个模板都用一个例子应用程序来评估其性能。表 5.3 列出了各种模板的例子。模板和对应的例子请访问相关网址下载安装[47]。例子保存在<template_name>/BF561/examples 中。这些例子在 ADSP-BF561 EZ-KIT Lite 评估板上测试过，同时需要摄像机源和输出视频输出（配置为 ITU-R-656 帧格式）。该例子处理的是 D1 尺寸（720×480）的图像。

表 5.3　每个模板的应用例子

模　　板	应　　用
行处理	颜色转换 YUV 4:2:2 到 RGB
宏块处理	边缘检测
帧内处理	运动检测

5.1.6 视频开发模板组合使用

有些应用程序可能具有多个图像处理算法，有不同的子处理块粒度，或者子处理块间有不同的数据依赖关系。此时，可以组合使用多个模板。以下给出几种组合使用的例子。

1．多个子处理块

有些应用可能基于扫描线进行预处理，然后再基于宏块进行处理。可以将扫描线模板和宏块处理模板组合成一个处理框架。此时，扫描线处理应该修改为将处理过的图像保存到 L2 或外部内存，然后使用宏块模板来处理保存好的图像。

2．对一个子处理块的多次访问

另一个例子，如直方图均衡，扫描线处理模板被使用两次。此时，图像要被访问两次。第一次是计算累计灰度值，第二次是应用均衡算法。对于该应用，L1 内存中每次处理一行，然后保存到外部内存，第二次处理时再由外部内存传输到 L1 内存中进行处理。处理结束后再保存到外部存储器，或直接输出到显示设备或保存成文件。

3．空域的数据依赖性

数据依赖性可能存在于图像的几行之间或宏块之间，因此可能需要同时访问多个子处理块。使用该模板的一个例子是运动估计算法，它要用到多个宏块。

4．双核/多处理器应用

这些模板可以组合起来为 ADSP-BF561 双核处理器（多处理器系统的一种）开发应用。基于双核编程模型，可以做到不同模板在不同内核上运行，能够提高并行运行效率，并使得系统资源达到均衡使用。

在资源有限的嵌入式平台上，有效地管理数据对于提高性能和改善系统带宽使用效率是至关重要的。上述基于 Blackfin 处理器的多媒体应用开发模板中采用了高效的数据管理技术，将帮助生成最优化数据布局，并加速开发。

5.2 Blackfin 处理器视频处理框架

在任何系统中，内存划分和数据流管理对于一个成功的多媒体框架设置至关重要[46]。Blackfin 处理器具有分层内存和非侵入式 DMA 和 PPI 接口。在应用程序中充分利用这些特性，可以提供更高的系统运行效率。为了在视频应用中获取 ADSP-BF533 和 ADSP-BF561 家族处理器的最大性能，需要具体分析以下问题。

1．内存使用原则

（1）内部内存空间。
（2）SDRAM 内存空间。
（3）管理外部数据访问。

2．PPI 采集和显示的 DMA 模式

（1）采用 ITU-R-656 输入模式。
（2）输出 ITU-R-656 视频帧。
（3）DMA 划分和流量控制寄存器。

5.2.1　内存使用原则

Blackfin 处理器结构支持分层内存，允许程序员让最经常使用的代码访问更快、更小的内存，视频数据缓冲区使用更大的内存。Blackfin 处理器内存具有统一的地址范围，包括内部 L1 内存、L2 内存（ADSP-BF561 处理器）、SDRAM 内存及异步内存空间。

内部内存空间 L1 内存运行在内核时钟频率，与其他内存相比延迟最低。此外，Blackfin 处理器具有分离的数据 L1 内存和指令 L1 内存空间。

L1 数据 SRAM 是由单端口构成的，每个子块包含 4K 字节内存。这样，在同时访问不同子块、或访问同一 4K 子块中一奇一偶的 32 位字时就呈现出多端口特性。如图 5.4 所示为内部内存中不同缓冲区未经优化时的内存分配。图中每块代表一个 4K 字节的内部内存子块。在这种分配方案下，内部数据总线没有得到充分利用，因为处理器不能同时拿到两个数据字。

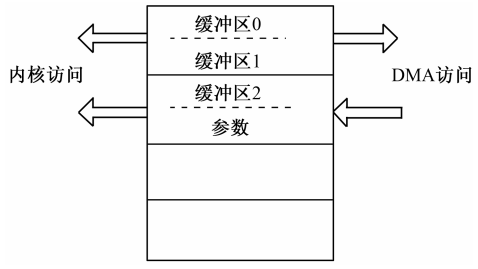


图 5.4　未经优化的 L1 内存分配

如图 5.5 所示为内部 4K 字节数据内存块优化后的内存分配。该内存分配允许同时进行双 DAG 和一个 DMA 访问，因此在数据总线上最大化了吞吐量。在视频编解码应用中，优化后的内存分配降低了访问 L1 数据内存的延迟，因为可以同时从内核和 DMA 控制器访问数据。

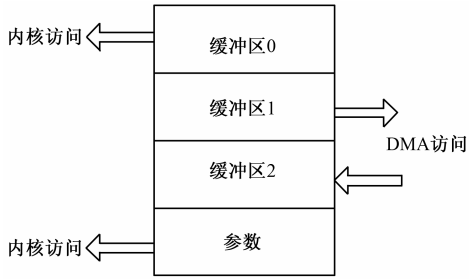


图 5.5　优化后的 L1 内存分配

SDRAM 控制器（SDC）使处理器能够与 SDRAM 间传输数据。SDRAM 控制器支持到 SDRAM 中 4 个内部子块的连接。在中断应用中，通过将数据缓冲区映射到不同的内部子块，可以最小化通过内核或 DMA 访问数据的延迟。SDC 能够同时跟踪每个子块的每一行（最高为 4 个内部 SDRAM 子块），因此能够随时在 4 个内部 SDRAM 子块间快速切换。

在图像处理应用中，数据帧通过 PPI 的 DMA 传入内存中。因为图像尺寸较大（如 VGA、D-1、4CIF、16CIF 等），图像的每一帧必须采集到 SDRAM 内存——使用 PPI 的 DMA 通道。算法可以从 SDRAM 中成块地读入像素数据，并依次处理每一块读入的数据。当内核在处理当前帧的缓冲区时，PPI 能够同时采集下一帧到另一个缓冲区中。因为内核和 DMA 可以同时访问 SDRAM 内存，必须要对代码、视频帧和其他缓冲区合理地进行映射，以最小化 SDRAM 内存访问相关的时延。

如图 5.6 所示为 SDRAM 内部子块未经优化的内存分配。在图 5.6 中，代码和视频数据都映射到 SDRAM 内部块 0。该分配方法会导致更大的时延，因为 SDRAM 行激活在每个时钟周期只能有一次。这是因为要切换内核访问（获取指令）和 DMA 访问到同一个 SDRAN 内部子块的不同的页。该延迟可能会导致 PPI FIFO 溢出错误（在图像采集中）或者下溢出（在图像显示中）。为了提高外部内存访问的吞吐量，必须合理分配视频缓冲区和参考缓冲区，以使得在任何时刻对一个 SDRAM 子块只有一个 DMA 访问。

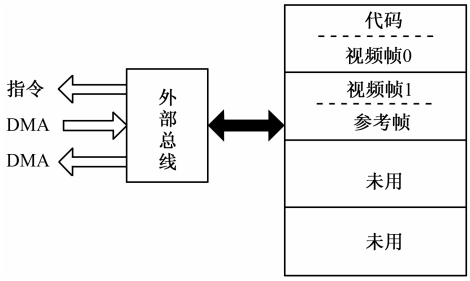


图 5.6　未经优化的 SDRAM 内存分配

如图 5.7 所示为优化后的 SDRAM 内部子块分配。在该内存分配例子中，在任何时刻，或者是内核、或者是 DMA 控制器来访问特定的内部 SDRAM 内存子块。因此，延迟达到了最小化，因为行激活周期分布到 SDRAM 访问的多个访问中。

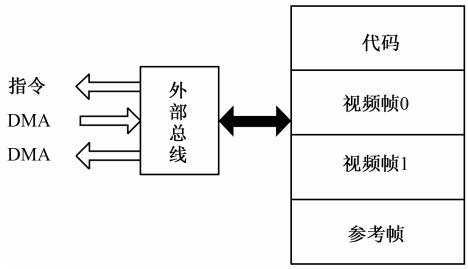


图 5.7　优化后的 SDRAM 内部子块分配

当传输是同一个方向时，对外部内存的访问会更有效。当访问 SDRAM 内存时，在一个方向上执行一组传输（避免经常性的方向变化）能够降低数据传输的延迟。在 DMA 控制器上经常性地改变方向会因为在读操作后执行写操作增加延迟。

5.2.2　PPI 采集和显示的 DMA 模式

Blackfin 的 DMA 控制器可以在内存和外设间高效地传输数据。设计者可以选择合适的 DMA 模式（如停止模式、自动缓冲区模式或基于描述子的 DMA 模式）来传输数据。而且，程序员可以通过选择合适的 DMA 通道来选择特定外设的 DMA 优先级。

Blackfin 处理器的 PPI 端口支持工业标准 ITU-R-656 模式和 GPIO 模式，以及各种内部或外部帧同步选项。利用 PPI 和相关的 DMA 模式，可以进行图像的连续采集和显示。程序员必须选择合适的 DMA 模式，这样才可以实时地、不丢帧地处理每一帧图像。在图像编码应用中，PPI 可以配置为描述子链模式，将图像采集到两个或多个缓冲区中。内核可以处理其中一个缓冲区，而 DMA 同时填充另一个缓冲区。必须确保内核和 DMA 控制器不会访问同一个 SDRAM 子块，正如前面讨论结果所表明的，如图 5.8 所示。

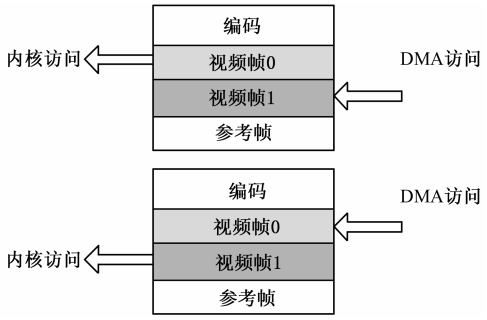


图 5.8　PPI DMA 和内核访问（在访问 SDRAM 内部子块时没有任何冲突）

在图像编解码应用中，处理器消耗的计算量（MIPS 值)是不固定的。消耗的 MIPS 值依赖于压缩率、采集的图像内容等。在图像解码应用中，如果要显示的解码帧还没有准备好，PPI 可以在此发送最近的解码帧。为获得该功能，PPI 可以配置为停止模式的 DMA。该模式对于视频显示具有更强的数据控制能力。在停止模式 DMA 中，每个工作单元完成后会产

生一个中断，此时 DMA 通道处于暂停状态。此时，如果下一帧显示数据没有准备好，刚才的一帧可以再次发送一遍。这可以在 PPI 的 DMA 中断子程序中实现。

1．使用 ITU-R-656 输入模式

对 ITU-R-656 帧数据，PPI 支持 3 种输入模式：全场模式、活动场模式、垂直消隐区域。在视频编码应用中，视频帧可以用活动场模式采集，这样只有场 1 和场 2 被采集。因为 ITU-R-656 采用交错式视频格式，视频算法可能需要视频数据必须是非交错格式的。使用内存 DMA，程序员可以直接完成对视频帧的解交错处理。

典型的 ITU-R-656 视频帧划分如图 5.9 所示。

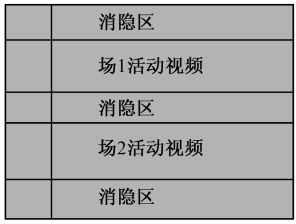


图 5.9　典型的 ITU-R-656 视频帧划分

为了最小化解交错帧数据的处理器开销，PPI 可以通过在每个活动行后跳过一行进行采集（如图 5.10 所示）。然后内存 DMA 通过填充消隐线将场 2 解交错到场 1。

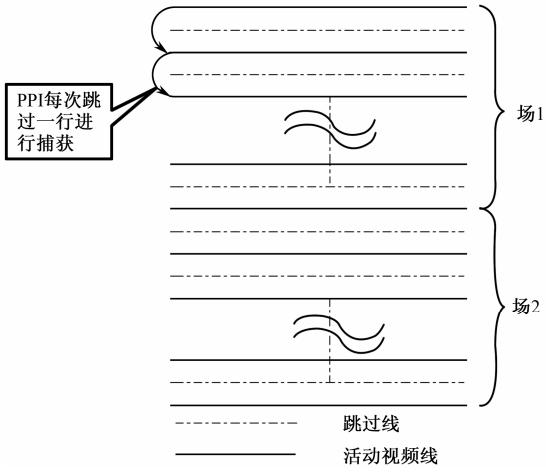


图 5.10　二维 DMA 采集交错线

如图 5.11 所示为使用内存 DMA 来解交错。场 2 中的数据必须与场 1 的数据解交错。因此，被跳过的扫描线被场 2 中的数据替代。因此，MDMA 源地址应该包含场 2 的第一行，而 MDMA 的目的地址应该包含第一个被跳过的扫描行。源和目的 MDMAx_Y_MODIFY 应该配置为跳过一行的数目。

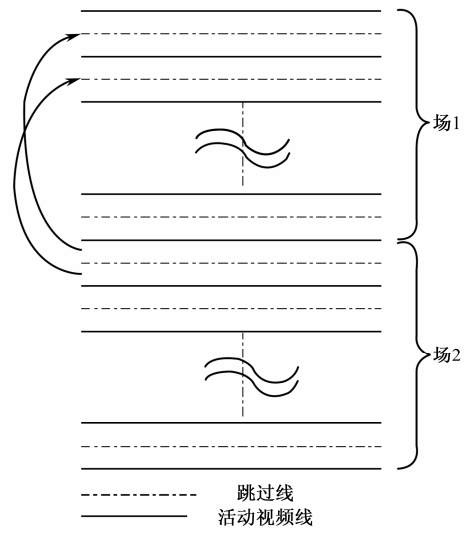


图 5.11　使用内存 DMA 解交错

2．输出 ITU-R-656 视频帧

PPI 并不显式地提供构建 ITU-R-656 输出流帧数据（具有正确的前导结构和消隐区间）的功能。然而，我们可以首先在内存中创建一个完整的帧，然后在 0 帧同步模式下通过 PPI 传输出去即可。视频数据、消隐数据和控制码可以提前在内存中设置好，然后再发送视频流。水平和垂直消隐信息可以在内存中（一次性地）建立起来，在帧与帧之间只需要更新活动场的数据，如图 5.12 所示。

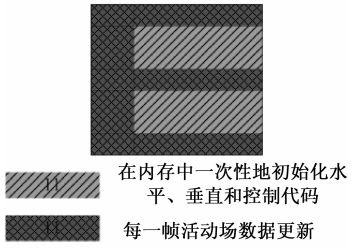


图 5.12　内存中的消隐及活动视频

3．DMA 划分和流量控制寄存器

在 Blackfin 处理器中，所有外设都支持 DMA。默认的，每个外设连接到一个特定的 DMA 通道。每个 DMA 通道具有本身访问内存的优先级。序号最小的 DMA 通道优先级最高。程序员可以将一个特定的 DMA 特定分配给该外设，这样就改变了外设 DMA 的优先级。

默认的，与其他外设相比，PPI 使用更高的优先级通道。如果一个应用程序有多个 DMA 并行运行，具有最高数据率或最小延迟需求的外设应该被分配更高的优先级（更小的序号），分配过程利用了 DMA_PERIPHERAL_MAP 寄存器。使用 DMA 流量控制寄存器，

程序员可以影响内部 DMA 总线（DAB、DCB、DEB）中数据移动的方向。流量控制提供了改变数据总线传输方向及改变频率的方法，实现方法是通过自动将同一方向的数据传输组织到一起。

DAB、DCB 和 DEB 总线为支持 DMA 的外设提供获得对片上、片外内存控制的方法，同时对内核到内存的带宽几乎没有影响。DMA 控制使用 DAB 总线来访问支持 DMA 的外设；DCB 总线用来访问内部内存；类似的，DEB 总线用来通过 EBIU 访问外部内存。

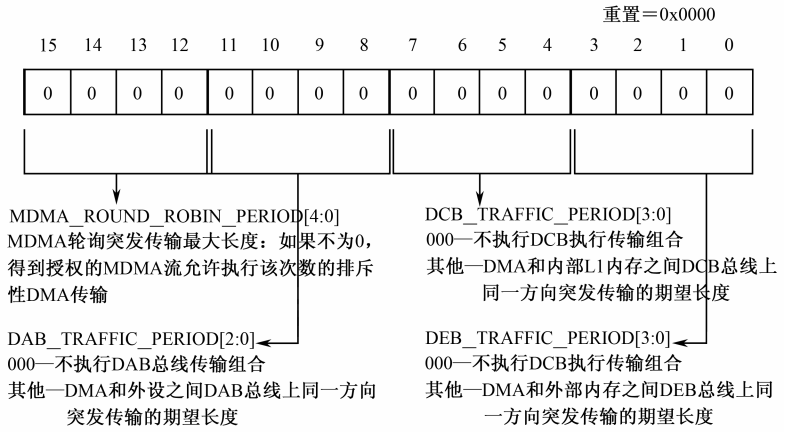


图 5.13　DMA 流量控制寄存器

通过私用流量控制寄存器，程序员可以独立地改变每个 DMA 总线的方向，即通过将相同方向的传输组织到一起。如图 5.13 所示为 DMA_TC_PER 寄存器的字段。例如，应用程序中 MDMA 和 PPI DMA 同时都是活动的，而且使用了流量控制寄存器。如果 PPI DMA 和 MDMA 都请求了 DEB 总线（MDMA 与流量相同而 PPI DMA 与流量相反），则 MDMA 被授予 DEB 总线，尽管 PPI DMA 具有更高的优先级。因为，PPI DMA 是与 DEB 总线流量相反的，PPI DMA 的有效优先级增加 16，而 MDMA 可以首先使用 DEB 总线。PPI DMA 则在流量控制计数器超时（或者流量停止，或改变方向）时获得 DEB 总线的访问权。

4．总线仲裁

当访问 L1 内存时，DMA 总线比内核总线有更高的优先级。默认的，当访问外部内存时，内核比 DMA 总线优先级高。通过设置 EBIU_AMGCTL 的 CDPRIO 位，所有的到外部总线的 DEB 间比内核访到外部内存的访问具有更高优先级。程序员可以根据具体应用的需求来使用该位。

5．DMA 和 Cache 的一致性

在应用程序中，如果内核和 DMA 同时访问共享缓冲区，并且 Cache 是使能状态，软件应该提供 Cache 一致性支持，即使共享缓冲区中的数据无效。在 Blackfin 处理器中，可以使用 MMR 寄存器来使其无效。而且，VisualDSP++开发工具提供了 C 库函数来使单个 Cache 子块无效。软件可以在访问共享的“易变”缓冲区前使 Cache 无效。

5.3　视频基础应用举例

本小节的讨论基于图 5.14 所示的视频处理流程示意图，包括了其中的各处理阶段。软硬件组件包括视频传感器、视频端口、解交错模块、扫描速率转换、媒体处理器、像素处理、色度信号下采样、伽玛矫正、YCrCb 到 RGB 的转换、尺寸缩放、显示处理及最后输出到 RGB LCD 平板。该系统仅仅是示例性的，一个特定系统并不需要配备上述所有这些元部件。此外，这些步骤的执行顺序也可能会发生变化。

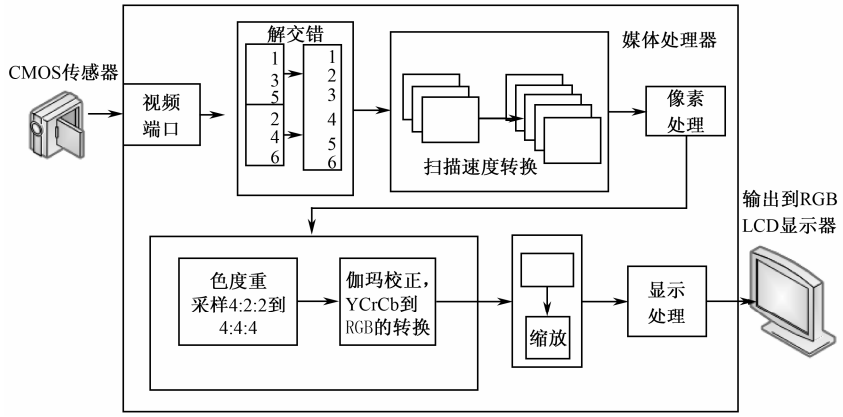


图 5.14　摄像机输入到 LCD 输出的数据流示例

下面我们简单介绍其中的一些基础处理。

5.3.1　解交错

将视频源数据从一个输出隔行 NTSC 数据的摄像机处取出时，往往需要对其进行去隔行处理，这样奇数行和偶数行将交织排列在存储器中，而不是分别位于两个分离的视场缓冲区中。去隔行不仅仅是高效率的基于块的视频处理所需要的，而且也是在逐行扫描格式中显示隔行视频所必需的（如在一个 LCD 平板上）。去隔行处理有多种方法，包括行倍增、行平均、中值滤波和运动补偿等。

5.3.2　解交错扫描速率转换

一旦视频完成了去隔行化，就有必要进行扫描速率的转换，以确保输入的帧速率与输出显示的刷新速率相匹配。为了实现两者的均衡化，可能需要丢弃场或者复制场。当然，与去隔行化类似的是，最好采用某种形式的滤波，以便消除由于造成突然的帧切换而出现的高频的人为干扰。

帧速率转换的一个具体情况，即将一个 24 帧/s 的视频流（通常是对应着 35mm 和

70mm 的电影录制），转换为 30 帧/s 的视频流，满足 NTSC 视频要求，这属于 3∶2 下拉（pulldown）。例如，如果每个胶片（帧）在 NTSC 视频系统中只被用到一次，则以 24fps 记录的电影的运动速度将提高 25%（＝30/24）。于是，3∶2 下拉被认为是一种让 24fps 视频流转换为 30fps 的视频序列的转换过程。它是通过以一定的周期化的样式重复各帧来实现的，如图 5.15 所示。

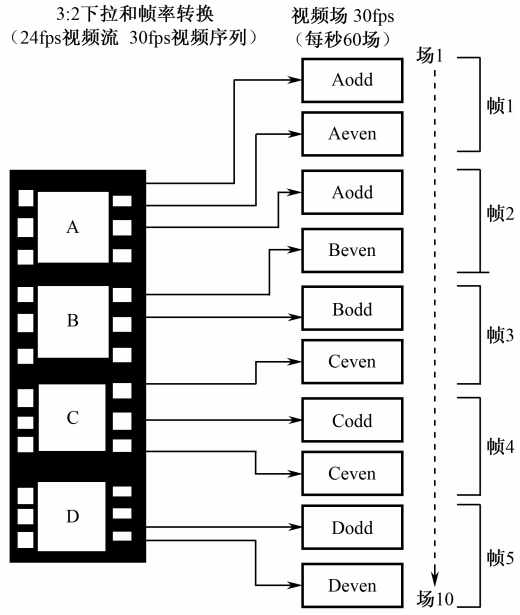


图 5.15　3∶2 下拉帧重复模式

5.3.3　像素处理

正如上面曾讨论过的那样，许多视频算法得到了广泛应用，它们可以被划分为空间和时间方面的诸多类别。一个极为常见的视频算子是二维卷积内核，它已经用于多种不同形式的图像滤波。

1．二维卷积

既然一个视频流事实上是一路以特定速率移动的图像序列，图像滤波器需要以足够快的速度运行，以跟上图像连续不断的输入。于是，重要的是，图像滤波器内核必须被优化，从而在尽可能少的处理器周期内完成运算。这可以通过考察基于二维卷积的简单图像滤波器组来予以演示说明。

卷积是图像处理中的基本操作。在二维卷积中，针对特定像素执行的计算是在其附近的像素点的亮度值的加权和。因为掩膜的周围都是以特定像素为中心的，故掩膜往往具有奇数次的阶数。相对于图像而言，掩膜尺寸一般很小，人们通常选择 3×3 掩膜，因为从逐个像素的角度出发，它在计算方面是合理的，同时足够大以能检测出图像的边缘。不过，

应该指出的是 5×5、7×7，甚至更多的像素组合，也得到了广泛应用。例如，摄像机的图像流水可以采用 11×11（甚至更大的内核），以便执行极为复杂的滤波操作。应当以一种对计算有帮助作用的方式来选择系数。例如，是 2 的幂次（包括分数）的缩放因子是首选方法，这样乘法就可以被替换为简单的移位操作。常用的卷积核包括 Sobel 边缘检测内核、平滑化内核、边缘增强内核等。

2．图像边界的处理

当一个诸如二维卷积的函数对图像边界附近的像素点进行操作时会发生什么情况？为了恰当地执行像素滤波，就需要这些边界“之外”的像素信息。出现这一情况，可以有两种补救方式。最简单的一种是，干脆忽略这些边缘区。例如，假定一个 5×5 的卷积内核需要分别选取当前像素的左、右、上、下的各两个像素，以保证运算的正常进行。于是，为何不在每个方向上删去两行，这样就可以保证该内核能始终对真实数据进行处理。当然，这并不始终是一种理想的方法，因为它丢弃了实际的图像数据。此外，当将滤波器串联起来，以形成更为复杂的像素处理能力时，随着每个新增加的滤波器级被执行，该方法将不断让输入图像幅面变窄。

处理图像在边界上出现的难题时，常用的其他方法是：复制多行和/或多列像素，或者从左边缘绕回前一图像的右边缘上，或是从底部边缘绕回顶部边缘处。虽然这在实践中可能易于实施，但它们也产生了此前并不存在的数据，因此在某种程度上会破坏滤波效果。

也许，最直接的，也是破坏最小的图像边界处理方法，是将处于实际图像之外的一切像素视为 0 值，或者说黑点。虽然这一方案也会造成滤波结果的失真，但它的侵入性并不像产生潜在随机的非 0 值像素的像素行时那么大。

5.3.4　色度再采样和颜色转换

我们的示例中的数据流最终需要转换到 RGB 空间中。在前面已经讨论了如何借助一个 3×3 的矩阵乘法在 YCbCr 4:4:4 和 RGB 空间之间进行相互转换。不过，到目前为止，我们的像素值依然处于 4:2:2 YCbCr 空间中。因此，需要对色度值进行再采样，以达到 4:4:4 的格式。接下来，到 RGB 的转换将是直接了当的，正如前面看到的那样。

从 4:2:2 通过再采样转换为 4:4:4 格式，需要从那些缺乏 Cb 和 Cr 分量之一的 Y 样本中插值出 Cb 和 Cr 值。一个简洁的再采样方法是从最邻近的像素上借助简单的平均化方法插值出缺失的色度值。也就是说，在一个像素点上缺失的 Cb 值将为最接近的两个 Cb 值的平均值所取代。某些应用还需要更高阶的滤波器，但这一简化的方法往往也已经够用了。另一种方法是对邻近的像素点的色度值进行复制，以得到这些在当前像素表示中缺失的量值。

一般来说，从 4:1:1 空间到 4:2:2 或 4:4:4 格式的转换只需用到一个一维的滤波器（拍数和数值应与所希望的滤波水平相一致）。不过，从 4:2:0 格式再采样为 4:2:2 或 4:4:4 格式，则还需要用到垂直采样，于是就有必要采用二维卷积内核。

由于色度的再采样和 YcbCr 到 RGB 的转换都是线性的运算，有可能将这些步骤组合起来，形成单个数学运算，从而高效率的实现 4:2:2 YCbCr 到 RGB 的转换。

5.3.5 缩放和裁切

视频缩放可以生成一路分辨率与输入格式的分辨率不同的输出流。理想情况下，固定缩放的要求（输入数据分辨率，输出平板显示的分辨率）都是事先已知的，以避免在输入和输出流之间进行任意的缩放所带来的计算上的负担。

缩放可以是缩减和放大，具体则取决于应用。清楚待缩放图像的内容（如文字和细线的存在与否）很重要。不恰当的缩放会导致文字不可阅读或者造成缩放后的图像中某些水平线的消失。

将输入帧尺寸的大小调整为幅面更小的输出帧时，最简单的方法是图像的裁切。例如，如果输入的帧尺寸是 720×480 像素，输出是 VGA 帧（640×480 像素），则可以把每行的前 40 和后 40 个像素丢弃。这样做的一个优点是像素的丢失和复制不会造成人为的影响。当然，缺点是将损失帧内 80 个像素（约 11%）的内容。有时，这并不是很大的问题，因为，由于显示器的机械外壳的存在，屏幕的最左侧和最右侧的部分（以及顶部和底部区域），往往成为人们无法看清的地方。

如果无法选用裁切的方法，还可以通过若干种方法来对图像进行下采样（减少像素和/或行的数量）或上采样（增加像素和/或行的数量），这使得人们可以在处理的复杂程度和相应的图像质量方面做出折中取舍。

1．增加或减少每行的像素数量

缩放方面所采用的一种直接的方法是丢弃像素（下采样）或者复制现有的像素（上采样）。也就是说，当缩减为一个较低的分辨率时，每条线（和/或每帧中的一定数量的线）的一些像素将被丢弃。这当然会减少处理的负担，但结果是造成了混叠和可见的人为影响。

只令复杂程度有少量上升的方法，是利用线性内插法来改进图像质量。例如，在缩减一幅图像时，在水平和垂直方向上的滤波可以获得一个新的输出像素点，该像素随后取代了在插值过程中所用的像素点。与前述技术类似的是，信息仍然会被丢弃，依然会出现人为的影响和混叠问题。

如果图像品质极为重要，还有其他方法可在不降低品质的前提下实现缩放。这些方法都是努力维持图像的高频分量，使之在水平和垂直缩放中保持一致，同时减少混叠的影响。例如，假设要求对一幅图像以 Y:X 为比例因子来缩放。为了达到这一目标，图像可以 Y 为因数进行上采样（“插值”），通过滤波以防止混叠，然后以比例因子 X 进行下采样（“抽样滤波”）。在实践中，这两个采样过程可以组合成为单个多比例滤波器。

2．增加或减少每帧的行数

增加或减少每行的像素点的指导原则，一般可以用于一幅图像每帧的行数的修改。例

如，每隔一行丢弃一行数据（或整个隔行场）提供了一种可减少垂直分辨率的快速方法。然而，正如我们以前提到过的那样，在丢弃或复制行时，应当采用某种垂直滤波方案，因为这些处理会在图像中引入人为影响。这里可以采用的同一种滤波策略是：简单垂直平均、更高阶的 FIR 滤波器或多比例滤波器，以便垂直方向上得到一个确切的缩放比率。

5.3.6　显示处理

1．Alpha 混合

在显示之前往往需要将两种图像和/或视频缓冲区组合起来。这样做的一个实用化的例子是蜂窝手机的图形显示上叠加的图标，如信号强度和电池电力指示等。一个涉及两路视频流的示例是“画中画”功能。

将两路流组合起来后，你需要决定在这些内容重叠的地方，哪路视频流将“胜出”。这就是 Alpha 混合发挥作用的地方。我们可以定义一个变量（α），它表示出如下在叠放流和背景流之间的“透明因子”：

输出值=α（前景显示的像素值）+（1–α）（背景像素值）。

该方程表明，α值为 0 时，可以实现完全透明的叠图，α值为 1 时，则叠图完全是不透明的，完全不显示相应区域的背景图像。

有时通过单独的通道与各像素的亮度和色度一起发送。这就造成了“4:2:2:4”的情形，其中最后一位值是一个伴随着每个 4:2:2 像素对象的α 键。α 的编码形式与亮度分量的编码相同，但对于大多数应用来说，常常只需取少数几个透明度的离散级别（也许是 16 个）即可。有时一个视频叠图缓冲区要预先乘以α，或者预先借助速查表进行映射。在这种情况下，它被称为一个“经过整形的”视频缓冲区。

2．合成操作

在合成操作中，需要定位一个叠图缓冲区到一个对应更大图像的缓冲区中。常见的一个实例是视频显示中的“画中画”模式，以及图形化图标（如电池和信号强度的指示标志）在背景图像或视频中的位置安排。一般来说，合成功能在输出图像完全完成之前尚需若干次反复循环。换句话说，产生一个复合的视频可能需要将“多层”图像和视频叠放起来。

二维的 DMA 功能对于合成功能来说是非常有用的，因为它容许在更大的缓冲区中定位出任意尺寸的矩形缓冲区。要记住的一点是，任何图像的裁切应该在合成过程之后进行，因为定位后的叠图可能会和此前裁切的边界发生冲突。当然，一个替代方法是确保叠图不至于和边界冲突，但有时这一要求太过于严格了。

3．色度键控

“色度键控”一词是指两幅图像合成时，其中一幅图像中的特定颜色（往往是蓝色或绿色）为另一幅图像中的内容所取代的现象。这提供了一种方便地将两幅视频图像合成起来

的方法，其方法是有意识的对第一幅图像进行剪裁，使之为第二幅图像的恰当区域所取代。色度键控可以在媒体处理器上以软件或者硬件形式来实现。

4．输出格式化

大多数针对消费类应用的彩色 LCD 显示（TFT LCD）都带有数字 RGB 接口。显示中的每个像素实际上都有 3 个子像素——每个都包含红、绿和蓝色滤波器——即人眼可以将其分辨为单色的像素。例如，一个 320×240 像素的显示事实上具有 960×240 像素分量，这包含了 R、G 和 B 子像素。每个子像素有 8 比特的亮度信号，这构成了常见的 24 比特的彩色 LCD 显示基础。

这 3 种最常见的配置中，要么是每个通道使用 8 比特来表示 RGB（RGB888 格式），要么是每通道 6 比特表示（RGB666 格式），或者 R 和 B 每通道用 5 比特表示，G 通道用 6 比特数据表示（RGB565 格式）。在这 3 种配置中，RGB888 提供了最好的色彩清晰度。这种格式总共有 24 比特的分辨率，可以提供超过 1600 万种色彩。它为 LCD TV 等高性能应用提供所需要的高分辨率和精度。

RGB666 格式在便携式电子产品中非常流行。这种格式具有总共 18 比特的分辨率，可以提供 262000 种色彩。不过由于其采用的 18 引脚（6+6+6）数据总线并不能很好地与 16 比特处理器数据通道相兼容，在工业上一种常见的折中方法是，R 和 B 通道 5 比特，G 通道 6 比特（5+6+5=16 比特总线），以此来实现与 RGB666 平板的连接。这种情形具有很好的性能，因为在视觉上绿色是 3 种颜色中最重要的。红色和蓝色通道的最低位连接到平板上对应的最高位上。这确保了每个颜色通道上的全动态范围（全亮一直低至全黑）。

对于视频处理其他的基本应用请参考相关资料[15,22]。

第 6 章　图像与视频处理软件开发包

为简化开发 ADI Blackfin 处理器应用程序的难度，提高学习效率及项目开发速度，ADI 公司提供了对应的软件开发工具包（SDK）[40]。该 SDK 为用户提供了可运行的系统级应用程序，展示了 ADI 提供的软件模块、系统服务和设备驱动程序。SDK 包括大量多媒体和音视频相关的应用程序，这些程序功能完整，支持各种常用设备。这个工具包的设计目标是加速开发过程、缩短学习曲线，以便更快地将产品推向市场。

开发包既有免费的应用软件及其代码，也包括一些实用工具、范例和文档等。利用 SDK 可以将其作为软件开发的基础框架，或者是作为学习如何使用 Blackfin 软件模块、系统服务和设备驱动程序的例子，或者是使用 Blackfin SDK 开发具有用户界面、图形覆盖、图像压缩与解压、流行编解码算法等的 Blackfin 可执行程序。这里所提供的用户应用程序是全功能的，并且能够免许可使用。其中有一些附带源代码，另一些只提供二进制的软件模块。此外还特别针对图像和视频处理等领域提供了专门的软件开发包，对用户开发多媒体应用提供了更大的便利。

本章首先介绍软件开发包的基本组成和使用方法，然后就图像处理开发包的使用给出几个具体的例子。其他开发包遵循相同的思路和框架，本章内容可以为学习、使用其他开发包提供帮助。

6.1　Blackfin 软件开发包介绍

为了测试、使用开发包，需要必要的硬件和软件工具。ADI 提供了两类初学者启动包：媒体播放器启动包和多媒体启动包。前者包括 ADSP-BF527/548 EZ-KIT Lite 评估板、评估版的 VisualDSP++，还有 SDK 光盘，其中有图像和视频处理、多媒体算法、图形库等例子程序。后者包括 ADSP-BF533/561 EZ-KIT Lite 评估板、音视频扩展板、USB-LAN 扩展板、评估版的 VisualDSP++及 SDK 光盘。当前 SDK 的版本是 4.0，需要下载 VisualDSP++5.0 Update 7 及以上的版本。

多媒体启动包中 SDK 包含以下内容：边缘检测、JPEG 编解码、MJPEG 编解码、Ogg Speex 编解码、简单的光栅图形包、图像视频处理工具、帧捕捉等。而媒体播放器启动包中包含：MP3 解码器、JPEG 解码器、AAC 解码器、文件系统、内存条、HDD 支持、SD 卡、MPEG-2 传输层解码器、视频输入/输出类驱动、音频编解码器输入/输出驱动、内存管理服务等。

6.1.1 SDK 的安装与使用

首先安装 VisualDSP++5.0 和 EZ-KIT Lite 软件。注意，要根据 SDK 的发布说明要求对 VisualDSP++进行更新。接下来安装 SDK，直接运行安装程序即可。注意，其默认安装路径是“%ADI_ROOT%\BlackfinSDK-X.00”，其中“%ADI_ROOT%\”代表 Analog Devices 程序的安装根路径，如“C:\Program Files\Analog Devices\”。

SDK 有以下子路径。

Algorithms（算法）：包含所有算法源代码及其工程构建文件。

Audio（音频）：包含构建和执行音频程序所需的所有源代码和工程文件。

BFinUtils：包含应用程序和用户需要的各种小工具源代码。

Bin：包含通用的基于 PC 的可执行程序，用来与 DSP 程序共同工作。

Documents（文档）：包含应用程序和算法相关的所有技术文档。

Include（包含文件）：包含应用程序需要的通用的包含文件。包含文件一般包含应用程序接口函数原型、外部引用和通用的宏定义。

Lib（库）：包含所有通用的预编译的库。有些库有 debug 和 release 版本，Debug 版本库名中有后缀“d”。有些库可由“Algorithm”路径下的项目重新生成。

Media（媒体）：包含多媒体或音频应用程序中使用到的所有媒体内容。

Multimedia（多媒体）：包含构建和运行多媒体应用程序所需的所有源代码和工程文件。该路径还包含 AVSync 目录。

Platforms（平台）：包含构建和执行多媒体播放器应用程序所需的所有源代码和工程文件。

Tools（工具）：包含 Blackfin USB 驱动器，用于将 PC 与开发硬件平台接口。

SDK 成功安装后，启动 VisualDSP++开发环境。VisualDSP++运行后，选择菜单项“File | Open | Project…”来打开希望的应用项目，或选择“File | Open | ProjectGroup…”来打开希望的项目组文件。例如，要打开项目“VideoInLcdOut”，操作如下：

（1）选择菜单“File | Open | Project…”，打开“Open Project”对话框。

（2）在“Look in:”下拉列表中设置路径为“%ADI_ROOT%\BlackfinSDK-X.0\Multimedia\VideoInLcdOut\ADSP-BF561”。

（3）从文件列表中选择“VideoInLcdOut.dpj”。

每个应用程序都有其“readme.txt”文件，打开该文件进一步了解以下指示：该应用程序需要的硬件；如何配置硬件；如何构建应用程序；如何运行应用程序。另外，该文件还概述了该应用程序所做的主要工作和工作方式。

6.1.2 SDK 中的应用简介

当前版本的 SDK 中包含以下应用程序。

（1）Audio_Player（音频播放器）：播放在 PC 上发现的特定编码的音频内容到扬声器。它需要一个主机程序来搜索并传输文件。支持的音频压缩格式为 AAC 和 MP3。

（2）FrameCapture（帧捕获）：从视频输入中捕获单帧图像，转换为 RGB565 格式，并存储到输出视频帧。

（3）FrameCaptureLcdOut：从视频输入中捕获单帧图像，转换为 RGB565 格式，并显示到 LCD 上。

（4）FrameCaptureVideoOut：从视频输入中捕获单帧图像，并显示到视频监视器，支持 NTSC 和 PAL 制。

（5）JPEGImageEdgeDetection：解压 JPEG 文件内容，并将检测到的边缘显示到视频显示器，支持 NTSC 和 PAL 制式。

（6）JPEG-MJPEG：解压在 PC 上发现的 M/JPEG 文件并显示到视频监视器。另外，它捕获视频输入，将其编码并保存到 PC 上。

（7）MediaPlayer：直接播放保存在（EZ-KIT Lite 支持的）存储设备上的媒体文件。图像会被显示到板上的 LCD 显示器，声音通过板上音频输出插孔输出。它是一个多线程的应用，使用内嵌的 RTOS（实时操作系统）——Micrium's uC/OS-II。支持的媒体格式包括 JPEG、MP3、AAC 和 WAV。调试输出通过 UART 发送出去，用户通过板上的键盘和拨轮输入。BF548 的板上存储包括硬盘驱动器（HDD）、SD 卡和 USB 存储棒。而 BF527 只支持 USB 存储棒。

（8）SensorCapture：从附带的 CMOS 传感器捕捉单帧图像，并显示到视频监视器或 LCD 显示设备，支持 NTSC、PAL 和 VGA 模式。

（9）SensorStream：从 CMOS 传感器捕获数据流，并输出到视频监视器或 VGS 显示器，支持 NTSC、PAL 和 VGA 模式。

（10）SpeexEcho：输入类似语音的数据流并编码成 Speex，然后将其解码并播放到扬声器。它还可以工作于“穿过”模式而不对数据流进行编码和解码。

（11）SRGP：显示了“简单光栅图形包（SRGP）”的 Blackfin 端口。

（12）VideoInEdgeDetection：在输入视频上进行边缘检测并将检测到的结果显示到视频监视器，支持 NTSC 和 PAL 显示模式。

（13）VideoInLcdOut：将视频输入数据转换为 RGB565，并在一个 LCD 监视器上播放，支持 NTSC 和 PAL 视频输入模式。

（14）VideoInVideoOut：来自视频捕获源的视频数据通过视频监视器绘制，支持 NTSC 和 PAL 视频输入模式。

（15）AVSync：将 MPEG-2 传输层流解复用为基本的 MPEG-4 视频流和 AAC 音频流，在同步的模式下播放这些基本流。

6.1.3　受限的软件

在该 SDK 中提供的一些软件算法是不完整的，但它对于工程开发和测试则是足够用的。

前十分钟，这些应用的运行毫无障碍，然后输出的结果会嵌入周期性的嘟嘟声或类似的奇怪现象。要获得该模块的产品版本，访问 www.analog.com/blackfinmodules。如果找不到相关的模块，还可以提交在线请求。

6.2 图形和视频处理软件开发包介绍

图像和视频处理是 Blackfin 处理器的主要目标应用之一，因此特别为图像和视频处理提供了特有的软件开发包。视频是由一系列帧图像组成的，因此图像处理构成了视频处理的基础，而视频开发则主要是解决运动目标检测、视频编解码等方面的技术问题。

6.2.1 图像处理开发包

Blackfin 图像处理开发包[42]是一组图像处理基本元素，用来帮助在 Blackfin 上快速开发复杂的图像或视频处理应用。它经过高度优化，可以运行于 ADI 的 Blackfin 处理器族。它是一个自包含的软件模块，兼容 MISRA-C。SDK 中还提供了相关的 C 参考代码和针对类 OpenCV[52]的封装代码，以及一些演示代码来示范如何在 Blackfin 上使用这些图像处理基元。这些代码能够处理在 L1 和 L3（外部内存，如 SDRAM/DDRAM)）内存中的图像。处理过程中，内存移动 API 用来帮助实现 L3 和 L1 内存之间的数据移动。

该开发包可以用于：视频分析，如目标、人脸检测，入侵检测等；汽车驾驶协助应用，如车道检测、交通信号识别等。

该开发包的特性包括：

（1）类 OpenCV 的 API。大多数模块与 OpenCV 完全相同（比特级别）。

（2）图像尺寸可配置。

（3）处理功能丰富，包括颜色转换、卷积、相关、Sobel 算子、形态学、Kalman 滤波、图像金字塔分解/滤波、矩阵操作、向量操作、积分图像、Hough 变换、光流计算、轮廓提取、Haar 特征提取和一些统计工具等。

（4）支持重入功能。

（5）输入格式可配置，包括 RGB888, YUV444 打包 YUV400 8 比特、YUV400 16 比特、HSV888、二进制 YUV 8 比特、二进制 YUV 16 比特等。

输出格式可配置，包括 RGB888, YUV444 打包 YUV400 8 比特、YUV400 16 比特、HSV888、二进制 YUV 8 比特、二进制 YUV 16 比特、矩阵/向量、灰度视频。

该开发包的下载和安装比较简单，默认安装路径为“%ADI_ROOT%\ SoftwareModules\ ImageProcessingToolbox-BF-Rel1.X.0_PROD”，输入文件路径为“%ADI_ROOT%\SoftwareModules\ Media\ImageProcessingToolbox-BF-Rel1.X.0_PROD\input”。

在“ImageProcessingToolbox-BF-Rel1.X.0_PROD”下，工程文件路径位于“demo\ adsp-bf5xx”，分别针对不同的平台；“libs”下面是库文件，其中包含了 Blackfin 图像处理

工具包库文件 libadi_image_tool_box.dlb 和用于划线等功能的图形库文件；“include”下是库文件对应的头文件；“c_ref”下包含了所有图像处理算法的参考代码；“cv_include”下是 OpenCV 库特定的头文件；“cv_src”下是 OpenCV 库特定的公共源文件；“demo”下面是对 OpenCV API 实现的封装，不同模块实现不同的功能。

安装路径下还有相关的文档，包括用户使用指导、产品说明、产品参考指导等，帮助用户更好地使用开发包。

6.2.2　视频处理开发包

视频处理开发包[43]是视频分析工具的一个实现，可以用于视频监控中事件的检测（如侵入检测、对象消失检测等）。当前版本中支持的是在静止摄像机拍摄的视频中检测前景对象。它可由 C 语言调用，使用灵活，可用于视频监控等场合，如安保摄像机、IP 摄像机等。

该开发包支持前景对象检测，同时能够指定感兴趣区域（ROI），实现只对感兴趣区域的检测。为了降低处理的计算量，视频输入可以在水平/垂直方向上缩小一半，以减小分辨率。其帧速率可以由用户指定，而且支持实现了帧级别的可充入，支持多通道处理。值得一提的是，它需要使用图像处理开发包中的相关内容。

开发包的下载和安装都比较简单，不在此过多介绍。其默认安装目录为“%ADI_ROOT%\SoftwareModules\VideoAnalyticsToolbox-BF-RelX.X.0_PROD\”。

接下来学习一些利用图像处理开发包开发应用程序的例子，视频处理开发包的介绍放到第 7 章来介绍。

6.3　Hough 变换及其实现

Hough 变换是一类非常典型的规则形状提取方法，它通过将边缘点映射到参数空间，再在参数空间中进行投票，得到对应的投票结果峰值，即可得到形状的参数。Hough 变换在目标提取、工业测量等方面有广泛的应用。本小节主要讨论直线段的提取方法。工具包中提供了 Hough 变换的 API 和例程，方便了开发者的使用。下面首先介绍 Hough 变换的基本原理，然后介绍其具体实现过程。

6.3.1　Hough 变换基本原理[10,12]

对于图像中某些符合参数模型的形状特征，如直线、圆、椭圆等，可以对其参数进行聚类，使用投票表决的方法提取相应的特征。最常用的表决方法之一是 Hough 变换。在 Hough 变换中，曲线上的每一点可以表决一组参数组合，赢得多数表决的参数就是胜者。下面介绍一下直线拟合数据的方法。直线方程是

$$y = mx + c \tag{6-1}$$

其中 x 和 y 是观测值，m 和 c 是直线的参数。如果已知参数值，则该点坐标之间的关系可以确定如下

$$c = y - mx \tag{6-2}$$

假定 m 和 c 是我们感兴趣的变量，而 x 和 y 是常数，则上述方程表示的是(m, c)空间中的一条直线，斜率和截距由 x 和 y 决定。点(x, y)对应于(m, c)空间上的直线，如图 6.1（a）、（b）所示。如果直线上有 N 个点，这些点就对应参数空间上一个直线族，且所有直线都经过该空间上某一点，该点坐标反映了(x, y)空间上的直线参数，如图 6.1（a）、（c）所示。

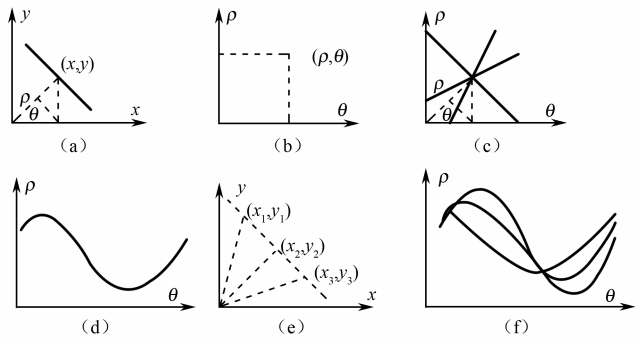


图 6.1　Hough 变换的原理

需要指出，参数空间曲线的形状取决于用于表示曲线的原始函数。实际中，常常使用直线的极坐标形式，而不是其显式形式，这样可以避免直线是垂直线时带来的问题。直线的极坐标方程表示如下。在直线坐标系中的直线 L，原点到直线的垂直距离为ρ，垂线与 x 轴的夹角为θ，则这条直线是唯一的，且其直线方程为

$$\rho = x\cos\theta + y\sin\theta \tag{6-3}$$

这条直线可以用(ρ, θ)的组合来唯一确定。该方程中，点(x, y)被映射为(ρ, θ)空间上的一条曲线（正弦和余弦曲线的加权和），如图 6.1（d）所示。如果直线上有 N 个点，那么这些点就对应(ρ, θ)空间上的 N 条正弦曲线，且曲线都经过(ρ, θ)空间上的某个点，如图 6.1（f）所示。该点的参数(ρ_0, θ_0)正是(x, y)空间上直线的参数值。这就是 Hough 变换检测直线的原理。对应其他曲线（如圆弧、椭圆），其参数个数可能不同，但是将坐标点(x, y)映射到参数空间，然后通过投票表决，同样可以实现对形状特征的提取。

利用 Hough 变换提取参数曲线时，通常把参数空间设计成一个累计器阵列，表示离散参数值。根据曲线方程，图像中的每一点对应参数空间的一组参数点，而参数空间（累计器矩阵）的峰值就表征一条曲线（如直线）的参数。具体实现算法如下：

（1）适当地量化参数空间，对于直线，参数空间取(ρ, θ)，量化后得到二维矩阵 $M(\rho, \theta)$。

（2）$M(\rho, \theta)$是一个累计器阵列，首先把累计器初始化为零。

（3）对图像空间中每个边缘点(x, y)，将θ 的所有量化值带入方程中，得到对应的ρ，并将 $M(\rho_i, \theta_i)$ 位置加 1。

（4）处理完所有边缘点后，分析 $M(\rho, \theta)$。如果 $M(\rho', \theta')$ 大于某个阈值，就认为存在一

条有意义的直线。

（5）由(ρ, θ)和(x, y)共同确定图像中的线段，并将断裂部分连接起来。

Hough 变换有显著的特点。它不需要预先组合或连接边缘点，位于感兴趣曲线上的边缘点可能构成图像边缘的一个小部分。而且，Hough 变换允许位于曲线上的边缘点存在一定程度的缺失。Hough 变换所基于的假设是在大量噪声出现的情况下，最好是在参数空间中去求满足图像边缘最大数量的那个点。因此 Hough 变换具有很好的抗噪性能。另外，如果参数空间的峰值包括了一个以上的累计器，则包含峰值的区域的形心就是参数的一个估值。

如果图像中有几条曲线和给定模型匹配，则参数空间中会出现多个峰值。此时，可以分析每一个峰值，去掉对应于某个峰值的曲线边缘，再检测余下的曲线，直到没有明显的边缘。但是，确定峰值的显著性总是存在一定的困难。

Hough 变换的另一个问题是离散参数空间随着参数的数量增加而迅速增加。例如，对于直线的情况，角度离散化的步长越小，提取的直线参数就越精确，但是同时累计器阵列则成倍增大，会影响计算效率。因此需要一个合适的折中，才能使其实用。

对一个圆弧段，参数空间是三维的，对于其他曲线，维数可能更高。由于累计器的数量随参数空间维数的增加而呈指数增加，Hough 变换可能对于复杂模型的计算效率很低。此时就需要对 Hough 变换算法进行改进。例如，对于圆的检测有 3 个参数，两个是圆心的坐标，另一个是圆的半径。如果可以得到边缘梯度角，就可以提供减少自由度数量的圆心坐标，从而得到所求的参数空间尺寸。从圆的中心到每一个边缘点的矢量方向由梯度角确定，剩下的位置参数只有圆的半径。其他一些减少参数空间的方法也在视觉中得到应用。

6.3.2　图像处理开发包中的 Hough 变换函数

用于 Hough 变换的 Blackfin 图像处理工具包内部 API 有如下 5 个：adi_HoughTransformInit、adi_HoughVoting、adi_FindLinesOfMaxLength、adi_FindLinesInRange 和 adi_HoughTransform。下面分别介绍这些函数，同时介绍其中用到的重要数据结构。

1．adi_HoughTransformInit

```
void adi_HoughTransformInit( const uint8_t *pInBuff, uint16_t dimY,
    uint16_t dimX, int16_t nThetaStart, int16_t nThetaEnd,
    uint16_t nRhoStep, uint16_t nThetaStep, int16_t *pSinBuff,
    int16_t *pCosBuff, int16_t nImageStride,
    ADI_HOUGH_DATA *pAdiHough );
```

这是 Hough 变换的初始化函数。所有的初始化函数保存在 ADI_HOUGH_DATA 结构中。使用者需要提供有效的缓冲区来保存 sine 和 cosine 值。该缓冲区的尺寸应该是[(nThetaEnd – nThetaStart)/ nThetaStep] * 2 字节。该函数用多项式近似填充 sine 和 consine 缓冲区，同时填充给定配置下期望的累计器尺寸 pAdiHough → nReqdAccSize=

[sqrt(dimX^2+dimY^2) + dimX] * (nThetaStar – nThetaEnd)/nThetaStep。

pInBuff 是输入图像缓冲区，图像应该是边缘图，作为二值图处理。

dimY 和 dimX 分别是图像/图像片的高度和宽度。

nThetaStart 和 nThetaEnd 确定了感兴趣线段角度范围，可以用它缩小搜索空间。

nRhoStep 是ρ 的步长，决定了累积数组的尺寸（成反比）。对整个范围要设为 1。nRhoStep 可以根据需要的精度改变。

nThetaStep 是角度的步长，决定着累积数组的尺寸。对于整个范围应设于 1，可以根据精度要求修改。增加 ThetaStep 会减小累计器尺寸，并提高投票速度，但是牺牲了精度。

pSinBuff 和 pCosBuff 分别保存 sine 和 cosine 值，这些值在投票时被使用。

nImageStride 是每行处理完后输入缓冲区指针增加量，可实现图像感兴趣区域（ROI）。

pAdiHough 是输出结构指针，调用者需要为其分配内存。该输出会用于后续的 APIs。

2．adi_HoughVoting

```
ADI_ITB_MODULE_STATUS adi_HoughVoting( ADI_HOUGH_DATA *pAdiHough,
    uint16_t *pAccumulator, uint32_t nAccumulatorSizePassed );
```

该函数填充累计器数组。对每个非 0 像素(x, y)，计算$\rho=x*\cos\theta+y*\sin\theta$ 并在(ρ, θ)处投票——对于累加器加 1。该过程要对所有非 0 像素及 nThetaStart 到 nThetaEnd 间的所有角度进行一遍（步长是 nThetaStep）。该函数假设第一次被调用前累计器被清空了。nAccumulatorSizePassed 不能小于初始化得到的最小内存。注意，应该首先调用 adi_HoughTransformInit 填充 ADI_HOUGH_DATA 结构。

pAccumulator 是累计器指针。nAccumulatorSizePassed 是传入累计器的尺寸，采用 16 位的半字。

3．adi_FindLinesOfMaxLength

```
int32_t adi_FindLinesOfMaxLength( ADI_HOUGH_DATA *pAdiHough,
    uint16_t *pAccumulator, uint16_t nMaxLinesToDetect,
    uint16_t nThreshold, int16_t *pDetectedLines,
    uint16_t nRhoNoise, uint16_t nThetaNoise );
```

对整幅图像完成投票后再调用本函数。本函数基于投票结果检测最多 nMaxLinesToDetect 条线段。该函数将用检测到的直线的ρ 和θ 值序列填充 pDetectedLines。投票结果必须大于 nThreshold 才能认为是条直线。这里，nRhoNoise 和 nThetaNoise 将设定期望直线的分辨率，即积分器平面的分辨率。相近的直线，即位于(nRhoNoise*nRhoStep)和(nThetaNoise*nThetaStep)范围内的认为是重复的。

该函数返回真正检测到的直线的数目。

4．adi_FindLinesInRange

```
void adi_FindLinesInRange( ADI_HOUGH_DATA *pAdiHough,
```

```
    uint16_t *pAccumulator, int16_t nRhoLow, int16_t nRhoHigh,
    int16_t nThetaLow, int16_t nThetaHigh, int16_t *pDetectedLines );
```

本函数必须在完成对整幅图像的投票后才能被调用。它将在给定区域内，根据投票结果，检测具有最大长度的单条直线。函数会用ρ和θ填充 pDetectedLines。区域是由ρ和θ范围确定的。注意，本函数值返回给定区域内具有最高投票值的一条直线。

nRhoLow 和 nRhoHigh 给出了ρ的上下界；nThetaLow 和 nThetaHigh 则该给出了θ值的上下界。pDetectedLines 中返回检测到的ρ和θ值。

5．adi_HoughTransform

```
ADI_ITB_MODULE_STATUS adi_HoughTransform( ADI_HOUGH_DATA *pAdiHough,
    uint16_t *pAccumulator, uint32_t nAccumulatorSizePassed,
    uint16_t nThreshold, uint16_t nMaxLines, int16_t *pDetectedLines,
    uint16_t nRhoNoise, uint16_t nThetaNoise,
    uint16_t *pActualDetectedLines );
```

本函数填充累计器数组，并找到最多 nMaxLines 条高投票值的直线。它实际上是融合了 adi_HoughVoting 和 adi_FindLinesOfMaxLength。本函数对整幅图像进行处理。这里 nRhoNoise 和 nThetaNoise 负责设置期望直线的分辨率。在映射空间相近，即(nRhoNoise*nRhoStep)和(nThetaNoise*nThetaStep)范围内的直线，被认为是重复的。本函数不支持基于分片的操作。在调用本函数前需要调用 adi_HoughTransformInit 来填充 ADI_HOUGH_DATA 结构。

nThreshold 是判断是否是直线的最小投票数；nMaxLines 是在一幅图像中要检测的最大直线数；pDetectedLines 保存检测到直线的 Rho 和 Theta 值序列。

6.3.3　基于图像处理开发包的实现

图像工具包附带的 Hough 变换例子工程名为“hough_transform_bf5xx.dpj”。其中将 Hough 直线提取功能封装到函数 ADIHoughTransformWrapper 中了。在主程序中，首先进行初始化，包括对图形库 GLUT 的初始化；然后读入图像文件内容到 L3 内存中，并调用 adi_GRAY2RGB 将数据从灰度转换为 RGB 格式；接下来调用 ADIHoughTransformWrapper 完成 Hough 变换的提取，得到直线的参数；最后还要调用 GetLineEndPoints 来确定直线的端点，并显示出来。

在 ADIHoughTransformWrapper 中，首先调用 adi_HoughTransformInit 完成 Hough 变换初始化工作，接着调用 adi_MemMoveInit 完成内存移动的初始化。内存数据移动为 ADI_DATA_DMA_FROM_TO_L3 模式，采用乒乓缓冲区结构，通过 DMA 方式完成数据从 L3 内存到 L1 内存的移动。

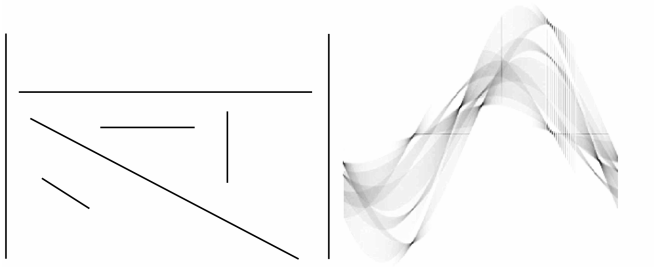
接下来，调用 adi_MemMoveIn 将一块图像数据读入 L1 内存中，通过调用 adi_HoughVoting 完成对这块数据的投票操作。该过程反复执行，直到文件结尾。项目中每

次读入 4 行，因此要求图像行数需要是 4 的倍数。

投票过程结束后，调用 adi_FindLinesOfMaxLength 来得到指定数目的直线的参数（ρ，θ）的数组。至此 ADIHoughTransformWrapper 函数结束并返回。

主程序中利用返回结果进一步调用 GetLineEndPoints 来确定每条直线的端点。该函数每次处理一条直线，根据输入的ρ和θ值，利用直线方程计算其端点。最后利用图形库将直线绘制出来。

如图 6.2 所示是一个 Hough 变换的具体例子，图 6.2（a）是原图，图 6.2（b）是 Hough 变换的累积结果。



（a）原图　　（b）Hough变换的累积结果（取反色）

图 6.2　Hough 变换的累积结果示例

6.4　腐蚀与膨胀运算的实现

6.4.1　形态学基本知识[10,12]

数学形态学形成于 1964 年，法国巴黎矿业学院 G. Matheron 和学生 J. Serra 从事铁矿核的定量岩石学分析时提出了该理论。数学形态学是一门建立在严格数学理论基础上的学科，其基本思想和方法对图像处理的理论和技术产生了重大的影响。目前，形态学图像处理已成为数字图像处理的一个主要研究领域。在文字识别、显微图像分析、医学图像、工业检测、机器人视觉等方面都有很成功的应用。

数学形态学是分析几何形状和结构的数学方法，它建立在集合代数的基础上，是用集合论方法定量描述目标几何结构的学科。这种结构表示的可以是分析对象的宏观性质，如在分析一个工具或印刷字符的形状时，研究的就是其宏观结构；也可以是微观性质，如在分析颗粒分布或由小的基元产生的纹理时，研究的便是微观结构。

形态学研究几何结构的基本思想是，利用一个结构元素（相当于模板）去探测一个图像，看是否能将这个结构元素很好地填放在图像的内部，同时验证填放结构元素的方法是否有效，如图 6.3 所示。对图像内适合放入结构元素的位置做标记，得到关于图像结构的信息。这些信息与结构元素的尺寸和形状都有关。构造不同的结构元素，便可完成不同的图像分析，得到不同的分析结果。

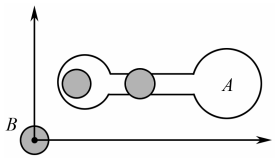


图 6.3　形态学研究几何结构的基本思想

1．数学形态学基本算法

设 A 和 B 为 R^2 的子集，A 为物体区域，B 为某种结构元素，则 B 结构单元对 A 的关系有以下 3 类：

（1）B 包含于 A：$B \subset A$

（2）B 击中 A：$B \bigcap A! = \varnothing$

（3）B 击不中 A：$B \bigcap A = \varnothing$

其示意图如图 6.4 所示。

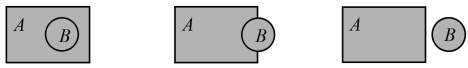


图 6.4　包含、击中和击不中示意图

2．腐蚀与膨胀

腐蚀表示用某种“探针”（即某种形状的基元或结构元素）对一个图像进行探测，以便找出在图像内部可以放下该基元的区域。定义如下：集合 A 被集合 B 腐蚀，表示为 $A\Theta B$，数学形式为

$$A\Theta B = \{x : B + x \subset A\} \tag{6-4}$$

它表示将 B 平移 x 但仍包含在 A 内的所有点的组成。若把 A 看做输入图像，B 看做模板，则 $A\Theta B$ 为在平移模板的过程中，所有可以添入 A 内部的模板的原点组成。

腐蚀过程示意图如图 6.5 所示。

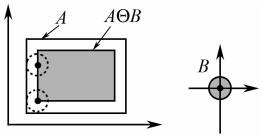


图 6.5　腐蚀过程示意图

膨胀是腐蚀运算的对偶运算，可以通过对补集的腐蚀来定义。A 被 B 膨胀表示为 $A \oplus B$，其定义为

$$A \oplus B = \left(A^C \Theta \left(-B\right)\right)^C \tag{6-5}$$

对于圆盘状结构元素，膨胀可以填充图像内部的小孔及在图像边缘处的小凹陷部分，并能够磨平图像向外的尖角。

膨胀过程示意图如图 6.6 所示。

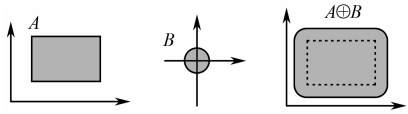


图 6.6　膨胀过程示意图

膨胀还可以通过相对结构元素的所有点平移输入图像，然后计算其并集得到。

$$A \oplus B = U\{A + b : b \in B\} \tag{6-6}$$

3．开运算和闭运算

利用图像 B 对图像 A 做开运算，用符号 $A \cdot B$ 表示，其定义为

$$A \cdot B = (A \Theta B) \oplus B \tag{6-7}$$

开运算过程示意图如图 6.7 所示。

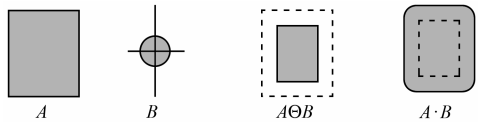


图 6.7　开运算过程示意图

闭运算是开运算的对偶运算，定义为先做膨胀运算然后再做腐蚀运算。利用图像 B 对图像 A 作闭运算表示为 $A \cdot B$，其定义为

$$A \cdot B = [A \oplus (-B)] \Theta (-B) \tag{6-8}$$

闭运算过程示意图如图 6.8 所示。

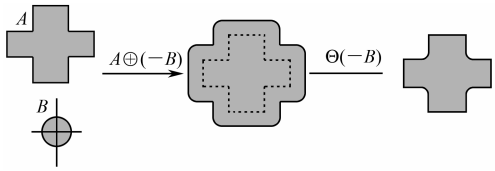


图 6.8　闭运算过程示意图

开运算可以滤掉背景（并）噪声，如椒盐噪声；闭运算可以滤掉前景（差）噪声，如沙眼噪声。

4．击中击不中变换

击中击不中变换需要两个结构基元 E 和 F，这两个基元被作为一个结构元素对 $B=(E, F)$，一个探测图像内部，一个探测图像外部，其定义为

$$A * B = (A \Theta E) \bigcap (A^C \Theta F) \tag{6-9}$$

显然：$E \bigcap F = \varnothing$

击中击不中变换示意图如图 6.9 所示。

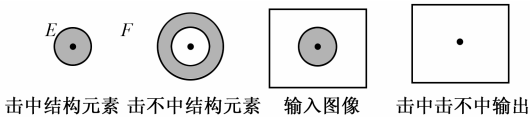


图 6.9　击中击不中变换示意图

可以利用击中击不中变换进行物体识别，如图 6.10 所示。

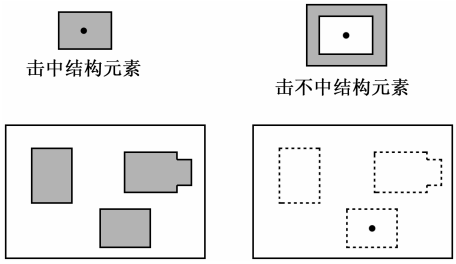


图 6.10　用击中击不中变换识别物体的过程

5．利用击中击不中变换细化

在实际应用中，通常选择一组结构元素对，迭代过程不断在这些结构对中循环，当一个完整的循环结束时，如果所得结果不再变化，则终止迭代过程。例如，如图 6.11 所示的是用于细化的 8 个方向结构对。

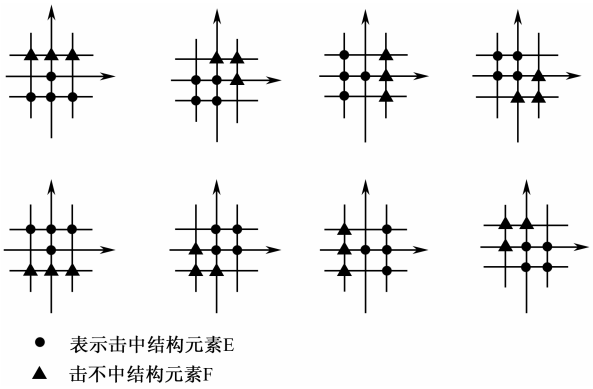


图 6.11　用于细化的 8 个方向结构对

6．形态学边缘检测

如图 6.12 所示，对于图像 A 和圆盘 B，$(A \oplus B) \setminus A$ 给出外边界，$A \setminus (A \Theta B)$ 给出内边界，$(A \oplus B) \setminus (A \Theta B)$ 给出跨骑在实际欧氏边界上的边界，又称形态学梯度。

7．骨架化处理

首先定义最大圆盘：对于一个目标 S，S 内的最大圆盘不是其他任何完全属于 S 的圆盘

子集，并且至少有两点与目标边界轮廓相切，如图 6.13 所示。

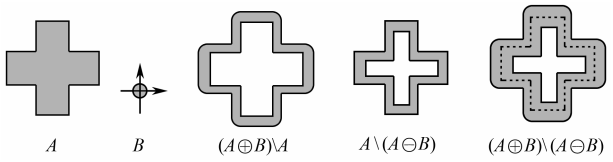


图 6.12　形态学边缘检测示意图

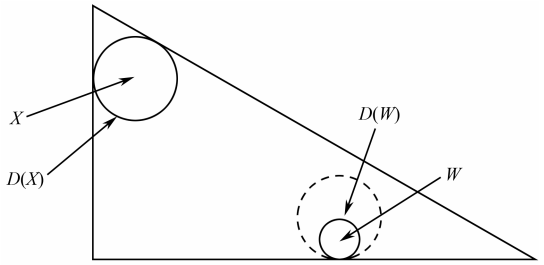


图 6.13　最大圆盘示意图

而骨架就是由所有最大圆盘的圆心构成的，如图 6.14 所示。

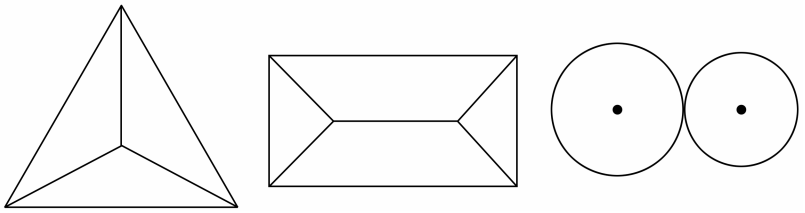


图 6.14　最大圆盘定义的骨架

数字情况下，令 B 为一种结构元素，最大圆盘可表示为 $0B$，$1B$，…，nB。其中

$$nB = \underbrace{B \oplus B \oplus \cdots \oplus B}_{n} \tag{6-10}$$

用邻接像素模板作结构元素时的数字圆盘如图 6.15 所示。

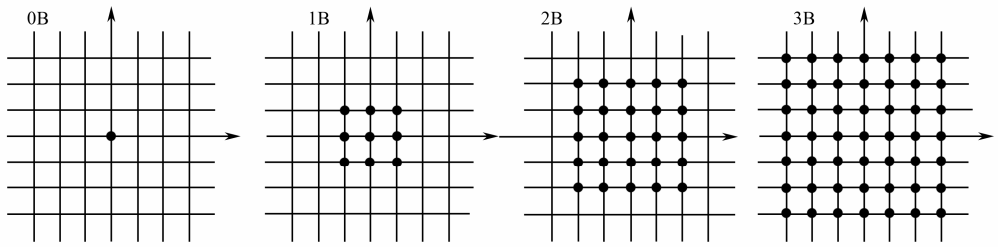


图 6.15　用邻接像素模板作结构元素时的数字圆盘

数字骨架的应用主要有数据压缩（有损或无损）和字符识别等。

除了以上介绍的内容外，形态学还包括其他丰富的内容，如流域分割法、灰度腐蚀与

膨胀、灰度开闭运算、Top-Hat 变换等。其应用领域包括图像滤波、图像细化和目标识别、焊点短路检测、粘连粒子分离、离散粒度分布统计和局部颗粒分析等。相关内容请参考形态学的相关书籍和资料[53~54]。

6.4.2　腐蚀与膨胀的开发包实现

Blackfin 图像处理工具包中腐蚀和膨胀运算的 API 分别是 adi_erosion 和 adi_dilation。

1．adi_erosion

```
void adi_erosion( const uint8_t *pInBuff, uint32_t dimY, uint32_t dimX,
    const int16_t *pMask, uint8_t *pOutBuff,
    ADI_MASK_OPTION eMaskOption );
```

该函数在输入灰度图像上进行腐蚀运算。eMaskOption 的有效选项是：ADI_MASK_CROSS 用于十字形掩膜；ADI_MASK_RECTANGLE 用于矩形掩膜；ADI_MASK_CUSTOM 用于任意形状的掩膜。这里我们假设，图像宽度必须是 4 的倍数，必须是字对齐的；输入图像和结构元素必须在不同的内存组中；因为 pInBuff 指向待处理的像素，希望在当前行的上一行有有效的像素，以确保像素访问有效性。调用该函数前，需要声明并建立 pInBuff 和 pOutBuff 对象。所有需要的内存块应该已经建立起来了。

pInBuff 指向输入图像/图像片。

dimY 和 dimX 分别代表输入图像/图像片的行数和列数。

pMask 指向 3×3 的结构元素。

pOutBuff 指向输出缓冲区。

2．adi_dilation

```
void adi_dilation( const uint8_t *pInBuff, uint32_t dimY, uint32_t dimX,
    const int16_t *pMask, uint8_t *pOutBuff,
    ADI_MASK_OPTION eMaskOption );
```

膨胀函数返回图像利用结构元素膨胀后的结果。该操作也称“填充”、“扩展”或“生长”。它可以用来填充形状上的“空洞”。该膨胀函数对输入的灰度图像执行膨胀运算。与前面一样，我们要求：输入图像宽度是 4 的倍数，且字对齐；输入图像和结构元素要位于不同的内存块上；在当前行的上一行要有有效的像素，以保证像素访问的有效性。pInBuff 和 pOutBuff 对象要提前声明并建立起来，所有内存必须在调用前完成。各个参数的意义与 adi_erosion 的完全类似。

6.5　人脸检测

基于图像和视频的人脸研究主要包括人脸识别和人脸检测。最初人脸研究集中在人脸

识别领域，并假设已经得到正面人脸或人脸很容易获得。近些年来，人脸检测作为独立的研究内容受到了普遍重视，在智能视频监控、基于内容检索、视频编解码、全新人机界面、电子商务等方面得到了广泛应用。

人脸检测指确定图像中人脸的数目、位置、尺寸、姿态等信息的过程，它是一个复杂的具有挑战性的模式检测问题。人脸作为一种模式，其特点是具有相当复杂的细节而且存在各种变化，如光照、表情、视角等都会改变人脸呈现出来的二维灰度分布结果。人脸目标检测的主要难点有两方面。一方面是由于人脸内在的变化所引起的：人脸细节复杂，变化丰富，如不同的表情如眼、嘴的开与闭等；人脸的遮挡，如眼镜、头发、胡须和头部饰物及其他外部物体等。另一方面是由于外在条件变化所引起的：由于成像角度的不同造成人脸的多姿态，如平面内旋转、深度旋转及上下旋转，其中深度旋转影响较大；光照的影响，如图像中的亮度、对比度的变化和阴影等；图像的成像条件，如摄像设备的焦距、成像距离、图像获得的途径等。这些困难都为解决人脸检测问题制造了难题。

对目标模式的识别归根结底要利用其不同于其他任何模式的一组特征。图像中人脸的特征非常丰富，如肤色特征、小波变换特征、特征脸空间分布、几何模板、轮廓、镶嵌图特征和结构特征等。如何选择显著特征并利用这些特征区分人脸与非人脸，是人脸检测要研究的关键问题。考虑到光照、表情、视角等变化都会导致人脸模式发生变化，一般需要采用多种模式特征综合的方法，如模糊决策、知识规则推理、Bayes 推理、统计推断、人工神经网络、支持向量机模型等方法。

传统人脸检测算法大体上分为人脸特征综合模型方法和基于统计模型的方法两大类。前者主要包括肤色区域分割与人脸验证方法和基于启发式模型的方法等，后者包括基于主元分析（PCA）、人工神经网络、概率模型和支持向量机等模型的方法。基于统计模型的方法更加流行，是解决复杂的人脸检测问题的有效途径。它具有如下优点：不依赖于人脸的先验知识和参数模型，可以避免不精确或不完整的知识造成的错误；采用实例学习的方法获取模型的参数，统计意义上更为可靠；通过增加学习的实例可以扩充检测模式的范围、提高检测系统的鲁棒性。人脸特征综合方法一般用于简单场景中的人脸检测，而基于统计模型的方法则大多适用于复杂背景图像中的人脸检测。

6.5.1 基于 Adaboost 学习的人脸检测[55]

研究者后来提出了一种基于 Adaboost 学习的人脸检测算法。基本思想是：挑选少量关键分类特征，构造出一组弱分类器，将其级联起来进行人脸检测。该算法速度近乎实时，检测结果也非常理想，是一种非常有前途的识别方法。对存在正面人脸的 384×288 像素大小的图像，在 700MHz Intel Pentium III 机器上，每秒可以检测 15 帧图像中的人脸，还可以借助序列图像差、颜色信息等进一步提高速度。超快速的人脸检测有许多实际应用，包括用户界面、图像数据库、远程会议等。速度上的提升将使得原来不够实用的系统能够成为实时人脸检测系统。在不需要快速检测的应用中，本算法则允许附加更多的后处理和分析。另外，它还可以进一步降低系统的整体功耗，便于在手持和嵌入式处理器上实现。ADI 的

图像处理包就提供了基于 Haar 特征的该类人脸检测的实现。

该算法主要分为 3 个步骤：计算积分图像、进行 Adaboost 训练、级联分离器。下面分别进行介绍。

1．矩形特征及积分图像

算法基于简单的图像特征，这些特征可以将难以由有限训练数据学习出来的先验知识编码到其中，而且比基于像素的系统更快。算法中利用了 3 类特征：双矩形特征、三矩形特征和四矩形特征，如图 6.16 所示。假设检测器基本分辨率是 24×24，所有可能的矩形特征有 160000 个，数目相当大，因此这组矩形特征是超完备的。

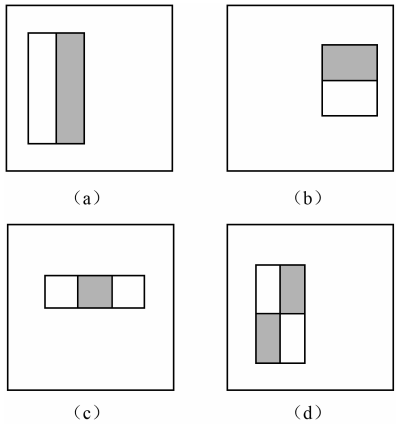


图 6.16　相对于检测矩形窗口的矩形特征举例

图中白色矩形区域内像素之和要减去黑色区域像素之和。双矩形特征的例子为（a）和（b），（c）是三矩形特征，（d）是四矩形特征。上述矩形特征可以利用积分图像实现快速计算。积分图像在(x, y)点的值就是其左上角所有像素的和，包括(x, y)在内。

$$ii(x,y)=\sum_{x'\leqslant x,\,y'\leqslant y} i(x',y') \tag{6-11}$$

这里 $ii(x, y)$是积分图像，$i\,(x, y)$是原始图像。利用下面这组循环，可以通过对原始图像的一次遍历完成积分图像的计算。

$$s(x,y)=s(x,y-1)+i(x,y) \tag{6-12}$$

$$ii(x,y)=ii(x-1,y)+s(x,y) \tag{6-13}$$

此处 $s(x, y)$是累积的行之和，$s(x,-1)=0$, $ii\,(-1, y)=0$

利用积分图像，任何矩形内部的像素之和可以利用 4 个积分数值来计算，如图 6.17 所示。这样，两个矩形和之差可用 4 个积分值来计算，如图 6.18 所示。又由于两个矩形相邻，所以可以用 6 个积分数值来计算，三矩形特征需要用 8 个，四矩形特征则需要 9 个积分值。

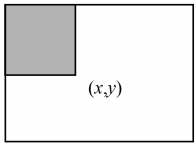


图 6.17　积分图像在点(*x, y*)处的值等于所有在其左上方像素值的和

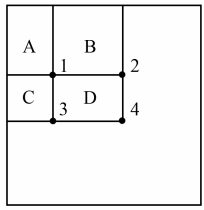


图 6.18　矩形 D 中的像素和可以利用 4 个数组值计算出来(4–1–(2+3))

2．学习分类器函数

给定一组特征集合及一组包含正负样本图像的训练集合，可以借助某种机器学习方法来学习分类函数。由于矩形特征数目庞大，即使每个特征可以高效计算，计算完整的特征集合的代价必然是昂贵的。从实验得知其中很小一部分特征可以组合成有效的分类器，主要的挑战是去找到这些特征。

AdaBoost 学习算法是用来增强简单学习算法的分类性能，它通过组合一组弱分类器函数来形成一个强分类器，简单学习算法称为弱分类器。为了增强弱学习器，反复调用 AdaBoost 来解决一系列的学习问题。在每轮学习之后，样本的权重会被调整，以强调那些被前一个弱分类器错误分类的样本。挑战在于将大的权值与好的分类函数关联，小的权值与差的分类函数关联。该学习算法被设计为：选择能够最好地区分正负样本的单个矩形特征。对于每个特征，弱分类器确定最优阈值分类函数，这样可以得到最少的错误分类比例。一个弱分类器 $h(x,f,p,\theta)$ 包含了一个特征 f，一个阈值 θ 和一个极性值 p，p 指示不等式的方向。

$$h(x,f,p,\theta)=\begin{cases}1 & \text{如果} pf(x)<p\theta \\ 0 & \text{其他}\end{cases} \tag{6-14}$$

这里 x 是图像的一个 24×24 的像素子窗口。实际上，没有任何单个特征可以以低概率完成分类任务。过程中，早期选择的特征产生 0.1～0.3 的出错概率，后期选择的特征，由于任务更加困难，产生 0.4～0.5 的出错概率。以下是该学习算法的步骤。

目标：构建 T 个弱分类器，每个使用一个特征。最终的分类器是 T 个弱分类器的加权线性组合，权值与训练误差成反比。

（1）给定样本图像 $((x_1,y_1),...,(x_n,y_n))$，这里 y_i 等于 0 和 1，分别对应负样本和正样本。

（2）初始化正负样本权值为 $w_{1,i}=\frac{1}{2m},\frac{1}{2n}$，$m$ 和 n 分别是负样本和正样本个数。

（3）对于 $t=1,\cdots,T$:

① 归一化权值。$w_{t,i}\leftarrow\frac{w_{t,i}}{\sum_{j=1}^{n}w_{t,j}}$

② 根据加权误差选择最佳弱分类器

$$\varepsilon_t = \min_{f,p,\theta} \sum_i w_i \left| h(x_i, f, p, \theta) - y_i \right|$$

③ 定义 $h_t(x) = h(x, f_t, p_t, \theta_t)$，其中 f_t、p_t和θ_t 使得以上误差取最小值。

④ 更新权值：$w_{t+1,i} = w_{t,i}\beta_t^{1-e_i}$。

如果 x_i 分类正确则 ε_i =0，否则 ε_i =1，且 $\beta_t = \dfrac{\varepsilon_t}{1-\varepsilon_t}$。

（4）最后的强分类器是

$$C(x) = \begin{cases} 1 & \sum_{t=1}^{T} \alpha_t h_t(x) \geqslant \dfrac{1}{2}\sum_{t=1}^{T} \alpha_t \\ 0 & \text{其他} \end{cases}$$

其中，$\alpha_t = \log(1/\beta_t)$。

为了最佳选择弱分类器，所有样本就某个特征按照特征值排序，对应的 AdaBoost 最优阈值可以通过一次扫描得到。对于列表中每个元素，需要保存和计算 4 个和：正样本权值 T_+之和，负样本权值 T_-之和，位于当前样本 S_+之下的正样本权值之和，位于当前样本 S_-之下的负样本权值之和。将排序队列中当前和前一个样本之间区间分开的阈值对应的误差是

$$\varepsilon = \min(S_+ + (T_- - S_-), S_- + (T_+ - S_+)) \qquad (6\text{-}15)$$

即将所有当前样本之下的样本标记为负和将所有当前样本之上的样本标记为正之间的最小误差（或相反方向）。搜索过程中这些和可以很容易进行更新。

实验显示由 200 个特征构造的分类器可得到理想的结果。对人脸检测，AdaBoost 选择的一些矩形特征是有意义的。第一个特征看起来是关注于眼睛区域通常比鼻子和下颌区域暗这样一个特性（如图 6.19 所示）。该特征相对于检测子窗口是比较大的，一定程度上对人脸的尺寸和位置不太敏感。第二个特征对应于眼睛比鼻梁暗这样一个特性。

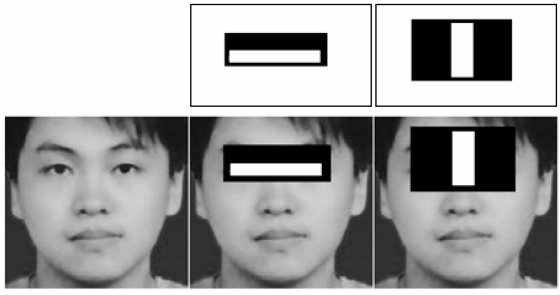


图 6.19　AdaBoost 选中的第一个和第二个特征

3．级联式分类器

进一步的，可以建立一个级联式分类器，以提高检测性能，并极大地减少计算量。目标是构建更小、更有效的增强分类器，快速拒绝非人脸的同时能检测出几乎所有人脸。简单分类器用来拒绝大多数子窗口，然后用复杂分类器来获得低的 FPR。级联中各阶段都利

用 AdaBoost 训练分类器来构建。从一个双特征强分类器开始，通过调整强分类器的阈值以最小化错误的负样本，可以得到有效的人脸滤波器。初始 AdaBoost 阈值为 $\frac{1}{2}\sum_{t=1}^{T}\alpha_t$，目的是保证分类器的低错误率。阈值越低，检测率和误检率就越高。该双特征分类器可以调整到检测 100%的人脸，而误检率为 50%，用很少的运算量显著减少了子窗口的数目。

检测过程的整体形式是一个退化的决策树，如图 6.20 所示。第一个分类器的正结果发送到第二个分类器，第二个分类器也调整到获得很高的检测率。第二个分类器的正结果再发送给第三个分类器，以此类推。在任何点的负结果则导致立即拒绝该子窗口。由于在任何图像内，大多数子窗口是非人脸的，级联器会在最初几步拒绝尽可能多的负样本。后面的分类器比前一个面临更困难的任务，需要复杂的特征，同时具有更高的误检率。

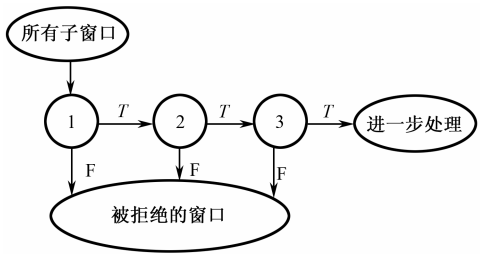


图 6.20　检测级联结构示意图

训练过程中，用户选择最大可接受 FPR 值和最小检测率。级联器每一层利用前述算法进行 AdaBoost 训练，使用的特征逐渐增加直到目标检测率和 FPR 满足要求。如果全局目标 FPR 不能满足，就再加一层。训练下一层所用的负样本是通过收集所有错误检测（通过在一组不包含任何人脸的图像上）运行目前的检测器。

6.5.2　基于图像处理开发包的人脸检测实现

利用 ADI 处理开发包中的 Haar 特征检测算法可以实现上述人脸检测算法。算法实现中直接采用了已有的训练结果，即不再重复训练过程。对应的项目名称是“haarfeatures_bf5xx.dpj”，主要处理过程如下（以图像文件输入为例）。

（1）将经过训练的模型参数读入 L3 缓冲区。

（2）将输入文件读入 L3 缓冲区，然后转换为灰度格式（调用 adi_RGB2GRAY）。

（3）调用 adi_HaarPreProcess 完成基于 Haar 特征对象检测的预处理，主要是完成积分图、平方积分图的计算。

（4）接着调用 adi_HaarFeaturesInit 完成 Haar 特征检测的初始化，主要任务是对 ADI_HAARCLASSIFIERCASCADE 结构的字段进行初始化。该结构包含了上述级联检测器的所有信息。

（5）调用 adi_HaarDetectObjects 进行 Haar 对象（如人脸）的检测。它多次对图像子窗口进行扫描，每次都要增加子窗口的尺寸，以便检测到所有尺寸的人脸。

（6）检测结束后，接着调用 adi_HaarPostProcess 完成 Haar 特征检测的后处理，主要任务是删除同一人脸在不同尺度和相邻位置的多次出现。

在整个处理过程中，结构 ADI_HAARCLASSIFIERCASCADE 起着重要作用。它是级联分类器对应的数据结构，定义如下。

```
typedef struct adi_HaarClassifierCascade {
    int32_t nCount;
    ADI_IMAGE_SIZE oOriginalWindowSize;
    ADI_IMAGE_SIZE oRealWindowSize;
    float32_t nScale;
    ADI_HAARSTAGECLASSIFIER *pStageClassifier;
    ADI_PVT_CLASSIFIERCASCADE *pPvtCascade;
} ADI_HAARCLASSIFIERCASCADE;
```

其中，nCount 是分类器级数（即“阶段分类器”的个数），oOriginalWindowSize 是子窗口的原始尺寸，oRealWindowSize 是当前子窗口的尺寸，nScale 是子窗口尺寸增加量，pStageClassifier 指向一组“阶段分类器”，pPvtCascade Pointer 指向一组伸缩了的级联分类器，它是内部私有的。

该结构在 adi_HaarFeaturesInit 中被初始化，在 HaarDetectObjects 调用中用于检测 Haar 特征。在评价子窗口中的对象是否是 Haar 特征时先调用 di_ReScaleFeaturesForSubWindow 函数，该函数按照当前窗口尺寸缩放级联分类器的参数，并按照缩放后的特征修改内部私有结构，同时还要对每个缩放后的特征修改积分图像指针。最后，在 adi_EvaluateSubWindow 调用中，该结构用于判断给定子窗口是否包含感兴趣的对象。以下是 adi_EvaluateSubWindow 函数的主要代码：

```
#pragma section("adi_fast_prio1_code")
int32_t adi_EvaluateSubWindow(
const ADI_HAARCLASSIFIERCASCADE *pCascadeClassifier,
const ADI_IMAGE_HAAR_DATA      *pImageData,
ADI_POINT                      oStartPoint,
int32_t                        nStartStage)
{
int32_t nReturnResult, nSumOffset, nSqSumOffset, i, j, nMean, nVariance;
int32_t nStageSum, nMeanSquare, nSum, nDifference, nResult, nIntegralSum;
int32_t nAccumSum, nThreshold, nLeftWeight, nRightWeight;
ADI_PVT_CLASSIFIERCASCADE  *pPvtClassifierCascade;
ADI_PVT_HAARCLASSIFIER    *pPvtClassifier;
ADI_PVT_HAARTREENODE     *pPvtNode;
nReturnResult = 1;
pPvtClassifierCascade = pCascadeClassifier->pPvtCascade;
```

```
nSumOffset=(oStartPoint.nY*pImageData->oImageSize.nWidth) + oStartPoint.nX;
nSqSumOffset=(oStartPoint.nY*pImageData->oImageSize.nWidth)+oStartPoint.nX;
// 计算积分结果
nIntegralSum = (pPvtClassifierCascade->p0[nSumOffset] - ( pPvtClassifierCascade
          -> p1[nSumOffset] + pPvtClassifierCascade->p2[nSumOffset]))
          + pPvtClassifierCascade->p3[nSumOffset];
nMean = adi_mult131_32bit(nIntegralSum,          // 计算均值
                          pPvtClassifierCascade->nInverseWindowArea);
// 计算方差——用于归一化
nVariance = ((pPvtClassifierCascade->pq0[nSqSumOffset] - (pPvtClassifierCascade
        ->pq1[nSqSumOffset] + pPvtClassifierCascade->pq2[nSqSumOffset]))
        + pPvtClassifierCascade->pq3[nSqSumOffset]);
nVariance = adi_mult131_32bit(nVariance,
                          pPvtClassifierCascade->nInverseWindowArea);
nVariance = adi_SqrRootFixed(nVariance, 16);
nSum = nVariance + nMean;
nDifference = nVariance - nMean;
nResult = adi_mult1616_1616(nSum, nDifference);
if (nResult > 0)  {   nVariance = adi_SqrRootFixed(nResult, 16); }
else              {   nVariance = ADI_ONE_IN1616;            }
for(i=nStartStage ; i<pPvtClassifierCascade->nCount ; i++)//逐级处理阶段分类器
{ nStageSum = 0;
    for(j = 0; j < pPvtClassifierCascade->pStageClassifier[i].nCount; j++)
{
      // 每个阶段分类器有多个 Haar 分类器
        pPvtClassifier = pPvtClassifierCascade->pStageClassifier[i].pClassifier + j;
        pPvtNode = pPvtClassifier->pNode;
        // 对阈值进行归一化
        nThreshold = adi_mult1616_1616(pPvtNode->nThreshold,    nVariance);
        // 计算积分和
        nIntegralSum = (pPvtNode->oFeature.oPvtHaarRect[0].p0[nSumOffset]
              - (pPvtNode->oFeature.oPvtHaarRect[0].p1[nSumOffset]
                + pPvtNode->oFeature.oPvtHaarRect[0].p2[nSumOffset]))
                + pPvtNode->oFeature.oPvtHaarRect[0].p3[nSumOffset];
        // 加权累积
        nAccumSum = adi_mult1616_32bit(nIntegralSum,
                          pPvtNode->oFeature.oPvtHaarRect[0].nWeight);
```

```
        nIntegralSum = (pPvtNode->oFeature.oPvtHaarRect[1].p0[nSumOffset]
              - (pPvtNode->oFeature.oPvtHaarRect[1].p1[nSumOffset]
                + pPvtNode->oFeature.oPvtHaarRect[1].p2[nSumOffset]))
                + pPvtNode->oFeature.oPvtHaarRect[1].p3[nSumOffset];
        nAccumSum += adi_mult1616_32bit(nIntegralSum,
                            pPvtNode->oFeature.oPvtHaarRect[1].nWeight);
        if(pPvtNode->oFeature.oPvtHaarRect[2].p0!=NULL){ //是否有第三个特征
            nIntegralSum = ( pPvtNode->oFeature.oPvtHaarRect [2].
                    p0[nSumOffset] - (pPvtNode->oFeature.oPvtHaarRect[2].
                    p1[nSumOffset] + pPvtNode->oFeature.oPvtHaarRect[2].
                    p2[nSumOffset]))+ pPvtNode->oFeature.oPvtHaarRect[2].
                    p3[nSumOffset];
            nAccumSum += adi_mult1616_32bit(nIntegralSum,
                            pPvtNode->oFeature.oPvtHaarRect[2].nWeight);
        }
        nLeftWeight = pPvtClassifier->pAlpha[0];     // 左权重、右权重
        nRightWeight = pPvtClassifier->pAlpha[1];
        // 对“阶段分类器”进行累积
        nStageSum+=(nAccumSum<nThreshold) ? (nLeftWeight):(nRightWeight);
    }
    if (nStageSum < pPvtClassifierCascade->pStageClassifier[i].nThreshold) {
        nReturnResult = -i;
        return (nReturnResult);
    }
}
   return (nReturnResult);
}
```

积分图像函数 adi_IntegralSum_i8（处理 8 位图像）代码如下：

```
void adi_IntegralSum_i8( const uint8_t   *pInBuff,   uint32_t    *pSum,
uint32_t dimY, uint32_t dimX, uint32_t nStride,
uint32_t *pRowVector, uint32_t       *pColVector )
{
uint32_t    i, j, nResult;
uint8_t     *pIn;
uint32_t    *pIntSum, *pRow, *pCol;
pIn = (uint8_t *)pInBuff; // 输入缓冲区指针
pIntSum = pSum;           // 积分和指针
```

```
pRow = pRowVector;          // 行向量指针
pCol = pColVector;          // 列向量指针
nResult = 0;
for (i = 0; i < dimY; i++) {
    // 第一列单独处理，因为要用到 I(x-1,y)
    nResult = *pRow;
    nResult = nResult - *pCol++;
    nResult = nResult + *pCol;
    nResult = nResult + *pIn++;
    *pIntSum = nResult;
    for (j = 0; j < (dimX - 1); j++) { // 列的中间和最后部分元素
        nResult = *pIn++;
        nResult = nResult + *pIntSum++;
        nResult = nResult - *pRow++;
        nResult = nResult + *pRow;
        *pIntSum = nResult;
    }
    pIntSum++;
    pIn = pIn + nStride;
    pRow = pIntSum - (dimX);
}
}
```

6.5.3 人脸跟踪算法的设计

在视频片段中，尽管上述人脸检测算法可以近乎实时地实现人脸检测，但还是仅限于几种训练过的人脸姿态。同时，由于存在各种干扰因素，如光照突变、瞬时遮挡等，算法无法保证在任何时刻都成功地检测到正确的人脸。为此可以在部分人脸成功检测帧的基础上，综合利用运动跟踪算法在其他帧中估计出面部运动轨迹。该跟踪算法基于低级图像特征，因而独立于人脸检测，对检测部分是有效的补充。另外，在许多应用场合我们还需要对人脸进行实时跟踪，如跟踪其运动轨迹，然后对其运动进行分类。

跟踪同样可以利用粒子滤波器实现，将其与特定的人脸重要特征相结合，能够获得更可靠的人脸检测结果，并在时间上实现平滑跟踪。在粒子滤波器算法的具体设计中，选择合理的人脸特征建立代表人脸特征的混合模型，是确保跟踪效果的关键。该模型既可包括直方图分布信息，也可包括边缘方向图分布。利用这些低级图像特征作为粒子滤波器中的相似度评价函数，即可实现对多目标的运动跟踪算法。

同时还可以在跟踪算法中，利用人脸检测结果对跟踪结果进行验证或修正。基本思想

如下：对检测与跟踪结果设置一个置信度，如果没有任何接下来的检测，置信度下降，被检测到则置信度上升。在跟踪过程中，如果置信度持续下降到低于某个阈值，则认为跟踪失败，需要重新进行人脸检测以便初始化置信度。采用了置信度的概念，就可用避免因为短时间内的遮挡等干扰造成跟踪失败。

6.6　图像处理软件包的内存使用

在上述图像处理应用的实现中，内存中数据的移动方式对性能影响显著。为此在此深入了解一下内存移动的相关知识。首先需要说以下 3 点问题。

（1）在 adi_MemMoveInit /adi_2DChainDesc_MemMoveInit 中，DMA 传输的单元是 16 位短字。

（2）内存分段，如表 6.1 所示。

表 6.1　图像处理内存分段表

段/用途	类　型	理想的内存位置	
		第一优先级	第二优先级
adi_fast_prio0_code	代码	片上 L1	L3 高速缓冲
adi_fast_prio1_code	代码	片上 L1	L3 高速缓冲
adi_fastb1_prio0_r	表	片上 L1A	L3 高速缓冲
adi_fastb0_prio0_temp	变量	片上 L1A	L3 高速缓冲
adi_fastb1_prio0_temp	变量	片上 L1B	L3 高速缓冲

使用过程中要注意，只要可能，所有代码和数据段都应该放到片上内存中。

如果处理器有足够多的 L2 内存，则 L2 内存可以作为 L3 内存的替代者。

（3）图像处理软件包用到的全局变量。图像处理库中使用了一些全局变量（表），表 6.2 中给出它们的名称、大小和位置。

表 6.2　工具包使用的全局变量

名　称	尺　寸	内　存　段
cDivisionTableRGBnHSV	512	adi_fastb1_prio0_r
rgbPixel	6	adi_fastb1_prio0_temp
cTable	36	adi_fastb1_prio0_r
cConstantTable	12	adi_fastb1_prio0_r
cMultCoeffYCbCr2RGB16to235	12	adi_fastb1_prio0_r
cFixedTable	40	adi_fastb1_prio0_r
cMultCoeffYCbCr2RGB0to255	16	adi_fastb1_prio0_r
cMultCoeffRGB2XYZ	18	adi_fastb1_prio0_r

续表

名　　称	尺　　寸	内　存　段
adi_temp_buff	48	adi_fastb1_prio0_temp
adi_morphology_left_buff	32	adi_fastb0_prio0_temp
Sobel5x5_8HorzCoef	48	adi_fastb1_prio0_r
Sobel5x5_8VertCoef	12	adi_fastb1_prio0_r
Sobel5x5_8CrossCoef	48	adi_fastb1_prio0_r
Sobel5x5_16HorzCoef	20	adi_fastb1_prio0_r
Sobel5x5_16VertCoef	20	adi_fastb1_prio0_r
Sobel5x5_16CrossCoef	16	adi_fastb1_prio0_r
anErrorDiffusionTempBuffer	4	adi_fastb1_prio0_temp
PolyCosCoef	20	adi_fastb1_prio0_r
sqrtcoef0	10	adi_fastb1_prio0_r
sqrtcoef1	10	adi_fastb1_prio0_r
pattern_arr_32_3I_5R	14	adi_fastb0_prio2_r
pattern_arr_32_4I_4R	14	adi_fastb0_prio2_r

6.6.1　内存移动流程

为了提高处理速度、测量机器周期数，图像处理操作需要处理 L1 中的数据。这样数据就需要在 L3 和 L1 间移动。为此，演示程序就可以通过一个封装来调用，而无需直接调用。

以下是封装代码为一维传输所做的事情，如图 6.21 所示。

（1）adi_MemMoveInit——选择传输模式。

① ADI_DATA_DIRECT_FROM_TO_L3：更新 L3 缓冲区指针。

② ADI_DATA_MEMCOPY_FROM_TO_L3：使用内存复制数据。

③ ADI_DATA_DMA_FROM_TO_L3：通过 DMA 传输数据。

（2）对每个片段（Tile）进行如下操作：

① 调用 adi_MemMoveIn 函数移入数据。

② 调用处理函数对移入的数据进行处理。

③ 调用 adi_MemMoveOut 函数移出数据（如果数据被修改了）。

（3）统计代码执行频率并打印出来。包括时钟数、处理占用的周期数、由模块和 DMA 占用的周期数等。

以下是封装代码为二维传输所做的事情，如图 6.22 所示。

（1）adi_2DChainDesc_MemMoveInit——选择传输模式。

ADI_DATA_DMA_FROM_TO_L3：通过 DMA 传输数据。

（2）对于每个片段，进行如下操作：

① 调用 adi_2DChainDesc_MemMoveIn 函数移入数据。

② 对移入的数据进行处理。

③ 调用 adi_2DChainDesc_MemMoveOut 函数移出数据（如果被修改了）。

（3）统计并显示代码执行频率。

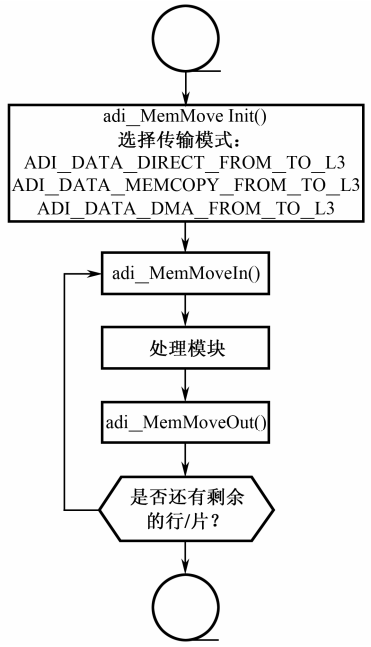


图 6.21　一维内存移动流程

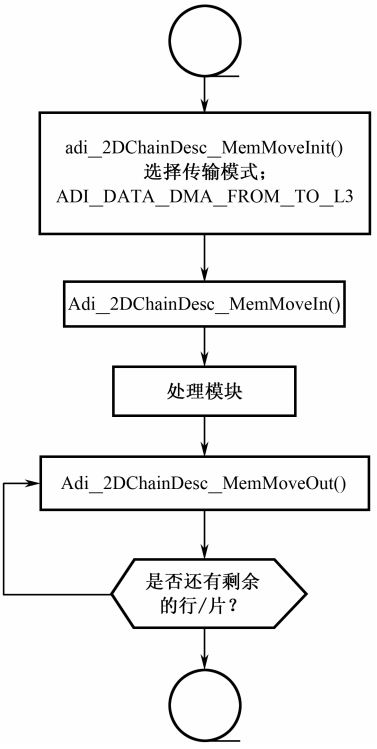


图 6.22　二维内存移动流程

6.6.2　一维内存移动 API

1．adi_MemMoveInit

```
int32_t adi_MemMoveInit( int8_t *pInputBufL3, int8_t *pOutputBufL3,
   uint32_t nInputBytes, uint16_t nSizeOfBuf, uint16_t nIncr,
    uint32_t nModeOfDataTransfer, int8_t **pOutputBufCurr );
```

该函数对内存移动相关变量进行初始化，同时设置传输模式。注意，pInputBufL3 和 pOutputBufL3 应该已经由调用者完成了内存分配。

参数中 pInputBufL3 指向输入 L3 缓冲区，pOutputBufL3 指向输出 L3 缓冲区，nInputBytes 为输入 L3 缓冲区中字节数。在 ADI_DATA_DMA_FROM_TO_L3 模式下，nSizeOfBuf 是第一个内存片段需要读入乒乓缓冲区的数据尺寸，nIncr 是每次数据传输后输入 L3 缓冲区指针的增量。nModeOfDataTransfer 给定内存移动的模式。pOutputBufCurr 指向输出缓冲区，其中转存图像处理的结果。

2．adi_MemMoveReset

```
void adi_MemMoveReset( int8_t *pInputBufL3, uint32_t nInputBytes,
    uint16_t nSizeOfBuf, uint16_t nIncr );
```

该函数重置内存移动中使用的全局变量。注意，pInputBufL3 应该已经建立起来。参数与上一 API 基本一致。

3．adi_MemMoveIn

```
int32_t adi_MemMoveIn( uint16_t nSizeOfBuf, uint16_t nIncr,
                       int8_t **pInputBufCurr );
```

该函数将需要的数据从 L3 读入内部内存。此前必须调用过 adi_MemMoveInit 完成初始化，并且已经设置了 gpInputBufL3 和 gpOutputBufL3 缓冲区指针。

参数中，nSizeOfBuf 是需要读入数据的字节数，nIncr 是输入 L3 缓冲区指针的增量，pInputBufCurr 指向需要进行处理的输入缓冲区的指针。

4．adi_MemMoveOut

```
void adi_MemMoveOut( uint16_t nSizeOfBuf, uint16_t nIncr,
                     int8_t **pOutputBufCurr );
```

该函数与 adi_MemMoveIn 作用相反，用来将处理过的数据转存到 L3 缓冲区。它也需要先调用 adi_MemMoveInit 完成初始化，并设置 gpInputBufL3 和 gpOutputBufL3 缓冲区指针。参数中 nSizeOfBuf 是要转存的字节数，pOutputBufCurr 指向输出缓冲区。

5．adi_MemMoveInEdge

```
void adi_MemMoveInEdge( uint32_t nFilterOrder, uint32_t dimX,
                        int8_t **pInputBufCurr, uint32_t nEdge );
```

该函数根据滤波器阶数读入边界数据——带多行重复的数据。需要重复的行数与滤波器阶数相关。对于 3x3 滤波器，只需要重复一行就可以构成 3 行作为一个片段：顶部边界处是第 1 行、第 1 行和第 2 行；底部边界处是第(N−1)行、第 N 行和第 N 行。而对于 5x5 滤波器，需要重复两行才行。

参数中 nFilterOrder 是滤波器阶数，dimX 是图像宽度，pInputBufCurr 指向需要处理的输入缓冲区，nEdge 指示是顶部边缘还是底部边缘。

6．adi_MemMoveValue

```
void adi_MemMoveValue( int8_t *pBuff, uint32_t nSizeOfBuf,
                       uint32_t nValue );
```

该函数将用户特定数值写入给定缓冲区。它也需要先调用 adi_MemMoveInit 完成初始

化。参数中，pBuff 指向数据要写入的缓冲区，nSizeOfBuf 是要写入数据的字节数，nValue 是要写入的值。

7．adi_MemMovePartialOut

```
void adi_MemMovePartialOut( uint16_t nSizeOfBuf, int16_t nModify,
                            int16_t nIncr, int8_t **pOutputBufCurr );
```

该函数将处理过的数据移动到 L3，其中 L1 中起始地址可配置。参数 nSizeOfBuf 是要移动到 L3 内存的字节数，nModify 指在移动到 L3 缓冲区之前 L1 缓冲区指针的增量，nIncr 则指移动后 L3 缓冲区指针的增量，pOutputBufCurr 是输出缓冲区。

8．adi_MemMoveModifyL3OutputBuffer

```
void adi_MemMoveModifyL3OutputBuffer( int8_t *pOutputBufL3,
                        uint32_t nIncr, int8_t **pOutputBufCurr );
```

该函数修改 L3 输出缓冲区变量，必须在 adi_MemMoveInit 之后调用。其中，pOutputBufL3 指向输出 L3 缓冲区，nIncr 是输出 L3 缓冲区指针的增量。

9．adi_MemMoveModifyL3InputBuffer

```
void adi_MemMoveModifyL3InputBuffer( int8_t *pInputBufL3,
                      uint32_t nInputBytes, uint32_t nIncr );
```

该函数修改 L3 输入缓冲区变量，必须在 adi_MemMoveInit 之后调用。其中，pInputBufL3 是输入 L3 缓冲区指针，nInputBytes 是输入缓冲区中的字节数，nIncr 是输入 L3 缓冲区指针的增量。

6.6.3　二维内存移动 API

1．二维内存移动数据结构 ADI_DMA_MEM_TRANSFER

```
typedef struct _adi_dma_mem_transfer {
    int8_t *pBaseBuffer;
    int32_t nIncrement;
    int32_t nPreBufferIncrement;
    int8_t bInputEnable2D;
    uint16_t nInputXCount;
    int16_t nInputXModify;
    uint16_t nInputYCount;
    int16_t nInputYModify;
    int8_t bOutputEnable2D;
    uint16_t nOutputXCount;
    int16_t nOutputXModify;
```

```
    uint16_t nOutputYCount;
    int16_t nOutputYModify;
    int8_t *pCurrBuffer;
    int8_t *pInOutBuffer;
} ADI_DMA_MEM_TRANSFER;
```

该结构作为参数传递给 2D DMA 函数，其中包括缓冲区参数。其中字段如下。

pBaseBuffer：指向 L3 缓冲区。

nIncrement：每次传输后 L3 缓冲区的增量。

nPreBufferIncrement：传输前 L1 缓冲区的增量。

bInputEnable2D：使能 2D 传输作为输入。

nInputXCount：一行中的输入字节数。

nInputXModify：DMA 传输中一个输入行的步长。

nInputYCount：一列中的输入字节数。

nInputYModify：DMA 传输中一个输入列的步长。

bOutputEnable2D：对于输出使能 2D 传输。

nOutputXCount：一行中输出字节数。

nOutputXModify：DMA 传输中一个输出行的步长。

nOutputYCount：一列中的输出字节数。

nOutputYModify：DMA 传输中一个输出列的步长。

pCurrBuffer：指向 L3 中的当前缓冲区位置。

pInOutBuffer：指向要使用的当前 L1 缓冲区。

2．adi_2DChainDesc_MemMoveInit

```
int32_t adi_2DChainDesc_MemMoveInit(
    ADI_DMA_MEM_TRANSFER **pInputBufL3,
    ADI_DMA_MEM_TRANSFER **pOutputBufL3,
    uint32_t nInput, uint32_t nOutput, uint32_t nModeOfDataTransfer );
```

该函数初始化 DMA 内部变量，并发出第一次传输。pInputBufL3 和 pOutputBufL3 要传递给后续的 DMA API。现在只支持 ADI_DATA_DMA_FROM_TO_L3 模式。DMA 被配置为一次传输读入一个 16 位字。

pInputBufL3 指向一组 ADI_DMA_MEM_TRANSFER 结构，定义了输入缓冲区。

pOutputBufL3 指向一组 ADI_DMA_MEM_TRANSFER 结构，定义输出缓冲区。

nInput 是输入个数，nOutput 是输出个数。

nModeOfDataTransfer 是传输模式。

3．adi_2DChainDesc_MemMoveIn

```
void adi_2DChainDesc_MemMoveIn(
```

```
ADI_DMA_MEM_TRANSFER **pInputBufL3, uint32_t nInput );
```

该函数启动下一组输入传输，并用当前要使用的输入缓冲区来填充 pInOutBuffer。pInputBufL3 即在 adi_2DChainDesc_MemMoveInit 得到的那个。

nInput 是输入个数。

4．adi_2DChainDesc_MemMoveOut

```
void adi_2DChainDesc_MemMoveOut(
    ADI_DMA_MEM_TRANSFER **pOutputBufL3, uint32_t nOutput );
```

该函数发起下一组输出传输，并用要使用的当前输出缓冲区填充 pInOutBuffer。pOutputBufL3 是在 adi_2DChainDesc_MemMoveInit 中得到的。

nOutput 是输出个数。

6.6.4　使用乒乓缓冲区进行内存移动

乒乓缓冲区用来通过 DMA 传输数据。以下两个场景解释了 DMA 传输过程。

1．场景一：单片段入单片段出

对于颜色转换和求均值，需要读入一个片段到 L1 进行处理，然后将处理过的片段转存到 L3。输入和输出片段不必尺寸相同。假设一个片段就是一行像素，那么要处理的输入数据就是一行图像，输出数据也是一行图像。如对于 adi_RGB2HSV：输入字节数=3(R, G, B)×图像宽度；输出字节数=3(H, S, V)×图像宽度。对于 adi_RGB2GRAY：输入字节数=3(R, G, B)×图像宽度；输出字节数=1(Gray)×图像宽度。

（1）调用 adi_MemMoveInit / adi_2DChainDesc_MemMoveInit。

① 设置数据传输模式为 ADI_DATA_DMA_FROM_TO_L3。

② 设置 L3 缓冲区地址的全局指针。

③ 重置缓冲区计数器。

④ 配置 DMA。

⑤ 初始化 DMA 基地址。

⑥ 设置输入和输出的 DMA 描述子。

⑦ 为“乒”缓冲区中第一个片段启动 DMA

（2）调用 adi_MemMoveIn / adi_2DChainDesc_MemMoveIn。

① 检查输入 L3 缓冲区中是否有数据，如果没有，等待前一个 DMA，并返回“乒”缓冲区指针。

② 否则，等待前一个 DMA 完成。

③ 为“乓”缓冲区启动 DMA。

④ 交换乒乓缓冲区指针。

⑤ 更新全局 L3 输入缓冲区计数器。

（3）图像处理。

① 现在输入数据位于 L1 中“乒”缓冲区。

② 处理该数据并转存到输出 L1 缓冲区。

（4）调用 adi_MemMoveOut / adi_2DChainDesc_MemMoveOut。

① 等待前一个 DMA 完成。

② 对于输出 L1 缓冲区中处理过 数据启动 DMA。

③ 更新全局 L3 输出缓冲区计数器。

单片段入单片段出示意图如图 6.23 所示。

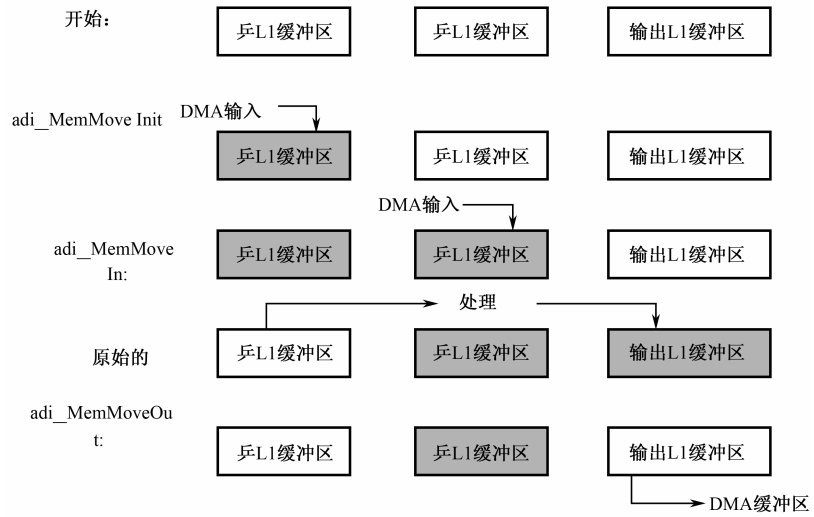


图 6.23　单片段入单片段出示意图

2．场景二：滤波器阶数片段入单片段出

出了场景一所述步骤，这里有两个新步骤。假设滤波器阶数=3，输入字节数=3(滤波器阶数)×图像宽度，输出字节数=1（处理过的行）×图像宽度。为了处理底部和底部边界，需要用最临近的像素对外围像素进行差值，这样就增加了两个额外的步骤。需要复制的像素数目依赖于滤波器阶数。

顶部位置处的 adi_MemMoveInEdge ，对 3x3 滤波器，只需要复制一行；对于 5x5 滤波器，需要复制两行数据。底部边界情况也是如此。

滤波器阶数片段入单片段出示意图如图 6.24 所示。

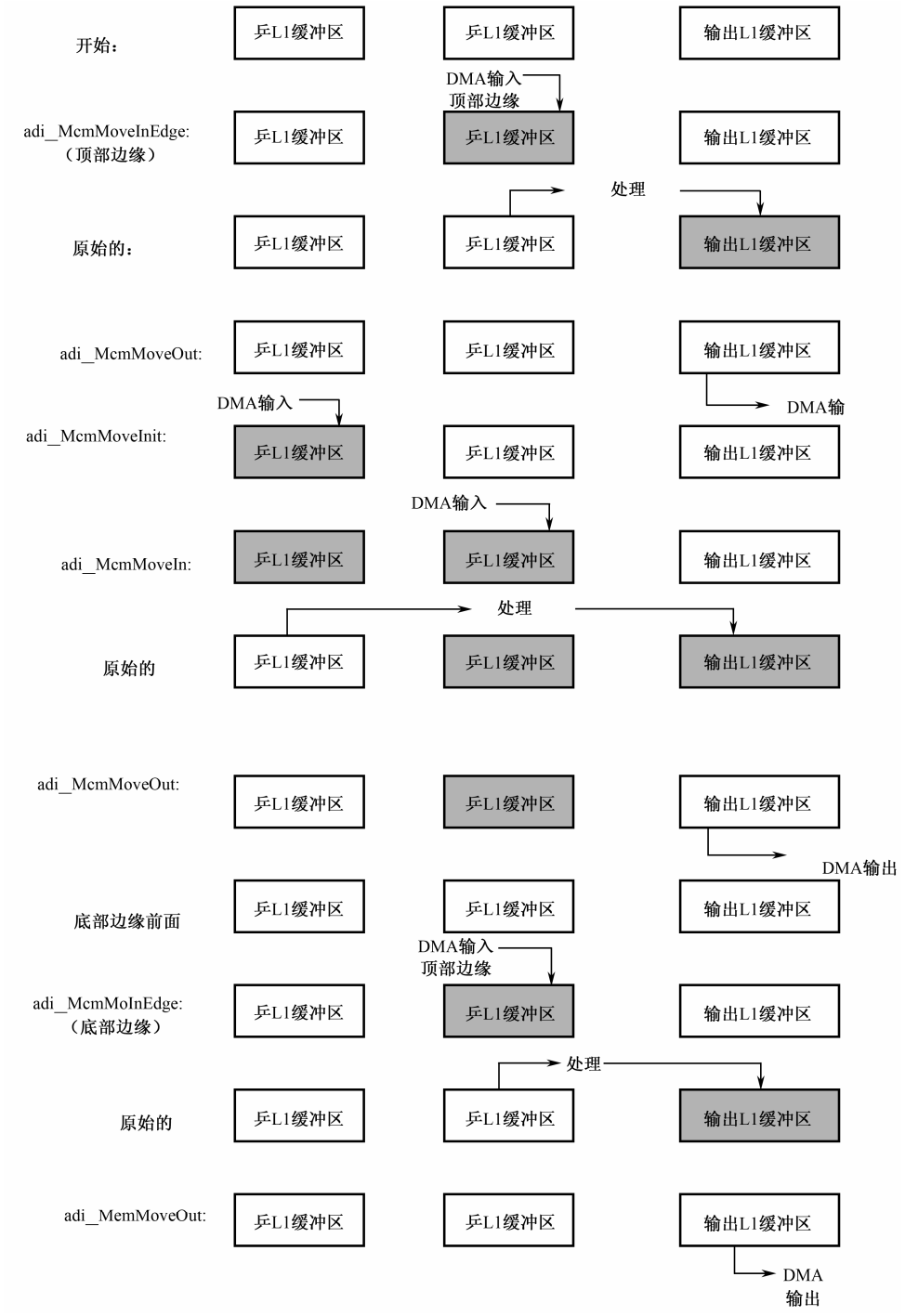


图 6.24　滤波器阶数片段入单片段出示意图

第 7 章　视频运动分析及应用

与静态图像处理相比，视频信号又引入了时间这一维度，能够记录一段时间内目标场景随时间的变化情况。与之相关的视频运动分析包含 3 部分内容：运动估算、运动分割和运动跟踪[13~17]。

运动估算又分为二维运动（图像平面运动）估算和三维运动（物体运动）估算。二维运动也称投影运动，它是三维运动在图像平面上的透视投影或正交投影。二维运动估算方法主要有光流分析法、基于块的分析法、像素递归法和贝叶斯法等。它能够为运动补偿、噪声滤波、数据压缩、目标检测、目标跟踪与识别等进一步处理提供更多的必要信息。

运动分割是目标跟踪和目标识别的基础。在静态图像中，哪个目标是运动的，或者说哪部分图像是目标，并不容易确定。在视频图像中，可以通过分析前后帧图像的变化来确定运动部分的图像或者目标。对于运动目标的检测，人们提出了许多方法。其中按照摄像头与背景之间是否有运动，分为静止背景和运动背景情况下两类方法。运动分割主要包括直接法、光流分割法和同时进行估算和分割的方法等。

一般来说，视频场景中目标的运动具有很强的连续性和规律性。而根据两个相邻帧的运动估算和分割结果，并不能充分利用这个重要信息来解决模糊度问题，如对于物体间瞬时的相互遮挡等。因此对于长序列的运动分析中应该充分利用运动跟踪技术，它将运动信息与一个描述实时运动进程的模型相结合，同时利用图像的局部灰度分布信息，可以有效地解决遮挡等模糊问题，同时极大地提高处理效率。运动跟踪采用的运动模型又分为二维轨迹模型和三维刚体运动模型，其典型算法包括 Kalman 滤波器和粒子滤波器等。

本章首先介绍这 3 方面内容的基本原理和技术，然后就其中典型技术的实现及 Blackfin 图像和视频处理开发包中的相关内容进行具体介绍，使读者熟悉视频运动分析应用开发的过程和算法实现基础。

7.1　运动估算

运动估算是视频运动分析的主要研究内容之一，它涉及二维运动（图像平面运动）或三维运动（物体运动）的估算。视频图像记录了场景中的物体在各时刻的二维投影图像，可以通过分析和处理视频系列图像，得到物体目标的运动变化信息。与静态图像比较，视频图像记录了目标物体的运动或变化过程的历史，通过时间轴上的相关性分析，可以获得比静态图像更多的目标信息。运动估算中一个主要的研究内容是二维运动估算，其主要目的是尽可能地把图像“结构化”，能够以对象为单位对图像进行描述和存储，为图像运动补

偿、图像噪声滤波、数据压缩、目标检测、目标跟踪与识别等进一步处理提供更多的必要的信息。现在大多数视频处理应用中都是对视频图像中的二维信息进行处理，因此本书主要介绍二维运动估算。

7.1.1　基于帧差的运动分析

本节介绍帧差模型，所有基于帧差的运动分析都是基于该模型基础上进行的。

图 7.1 表示了两帧图像进行差分的情形。一般情况下，图像可以分成目标和背景两大部分。目标可以有多个，图中只画出了一个。当目标和背景运动时，目标和背景的相对位置和形状就会发生改变。在两帧图像做差时，可以把图像分成几个不同的区域。“目标重叠区域”表示两帧中都有目标内容的区域，“目标覆盖区域”表示该区域在上一帧为背景而本帧为目标的区域，“目标暴露区域”表示上一帧为目标而本帧为背景的区域，“背景重叠区域”则表示两帧都为背景的区域。当然也有“背景覆盖区域”和“背景暴露区域”，由于都在图像边缘，位置固定，所以不必单独详细讨论。

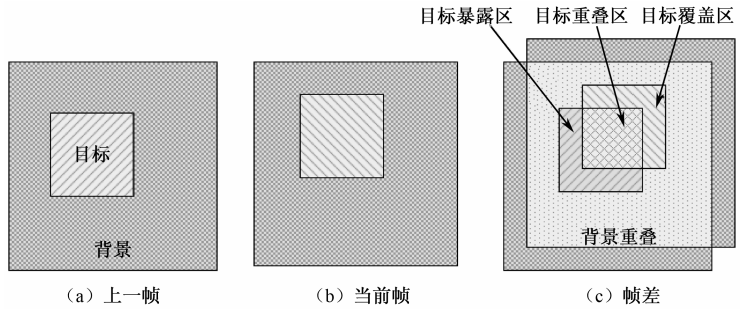


图 7.1　相邻帧目标变化示意图

对运动目标检测问题，我们主要讨论“目标重叠区域”、“目标覆盖区域”和“目标暴露区域”的情况。在图中，设目标以速度 $\boldsymbol{v}_m$ 运动，背景以速度 $\boldsymbol{v}_b$ 运动，它们都是时间 t 的函数。同时，由于只考虑平移运动，图像分布函数的形式不变，只是在位置上发生了变化，在数学函数上表现为位置变量随时间变化。设 $\boldsymbol{p}_o$ 表示原始起点位置，目标图像可以表示为式（7-1），背景图象可以表示为式（7-2）。经过时间Δt 后，目标和背景图像可以分别表示为式（7-3）和式（7-4）。公式（7-1）和式（7-2）中目标图像是覆盖在背景上的，两者不能在同一区域同时出现。位置 $\boldsymbol{p}_o$ 和 $\boldsymbol{v}_m$ 、 $\boldsymbol{v}_b$ 都是向量，具有方向属性。

$$f_m(\boldsymbol{p},t)=f_m(\boldsymbol{p}_o+\boldsymbol{v}_m t) \tag{7-1}$$

$$f_b(\boldsymbol{p},t)=f_b(\boldsymbol{p}_o+\boldsymbol{v}_b t) \tag{7-2}$$

$$f_m(\boldsymbol{p},t+\Delta t)=f_m(\boldsymbol{p}_o+\boldsymbol{v}_m(t+\Delta t)) \tag{7-3}$$

$$f_b(\boldsymbol{p},t+\Delta t)=f_b(\boldsymbol{p}_o+\boldsymbol{v}_b(t+\Delta t)) \tag{7-4}$$

当（a）、（b）两帧图像相减时，其结果在不同的区域，表示式不同。在目标重叠区域的结果式（7-5）表示，在目标覆盖区域的结果由式（7-6）表示，在目标暴露区域的结果由

式（7-7）表示，在背景重叠区的结果由式（7-8）表示。在这些表达式中设背景和目标图像函数的一阶导数处处存在。

$$\begin{aligned}\Delta f_{mm}(\boldsymbol{p},t+\Delta t) &= f_m(\boldsymbol{p}_o+\boldsymbol{v}_m(t+\Delta t))-f_m(\boldsymbol{p}_o+\boldsymbol{v}_m t)\\ &= f_m'(\boldsymbol{p}_o+\boldsymbol{v}_m t)\cdot(\boldsymbol{v}_m\cdot\Delta t);\end{aligned} \tag{7-5}$$

$$\begin{aligned}\Delta f_{mb}(\boldsymbol{p},t+\Delta t) &= f_m(\boldsymbol{p}_o+\boldsymbol{v}_m(t+\Delta t))-f_b(\boldsymbol{p}_o+\boldsymbol{v}_b t)\\ &= f_m'(\boldsymbol{p}_o+\boldsymbol{v}_b t)\cdot(\boldsymbol{v}_m\cdot\Delta t)+f_m(\boldsymbol{p}_o+\boldsymbol{v}_m t)-f_b(\boldsymbol{p}_o+\boldsymbol{v}_b t);\end{aligned} \tag{7-6}$$

$$\begin{aligned}\Delta f_{bm}(\boldsymbol{p},t+\Delta t) &= f_b(\boldsymbol{p}_o+\boldsymbol{v}_m(t+\Delta t))-f_m(\boldsymbol{p}_o+\boldsymbol{v}_m t)\\ &= f_b'(\boldsymbol{p}_o+\boldsymbol{v}_b t)\cdot(\boldsymbol{v}_b\cdot\Delta t)+f_b(\boldsymbol{p}_o+\boldsymbol{v}_b t)-f_m(\boldsymbol{p}_o+\boldsymbol{v}_m t);\end{aligned} \tag{7-7}$$

$$\begin{aligned}\Delta f_{bb}(\boldsymbol{p},t+\Delta t) &= f_b(\boldsymbol{p}_o+\boldsymbol{v}_b(t+\Delta t))-f_b(\boldsymbol{p}_o+\boldsymbol{v}_b t)\\ &= f_b'(\boldsymbol{p}_o+\boldsymbol{v}_b t)\cdot(\boldsymbol{v}_b\cdot\Delta t);\end{aligned} \tag{7-8}$$

由此可以看出，在目标重叠区域和背景重叠区域，差分结果就是各自运动图像的一阶微分与运动速度和时间间隔的乘积。在图像上表现为边缘增强运算。在图像各区域的边界上，出现较大的数值，在灰度缓变区，出现很小的数值。

在目标覆盖区域和目标暴露区域，除了微分外，还有目标和背景的差值。只有目标和背景的运动速度（包括方向）完全相同，而且在交界处目标和背景的取值（或极限）相等，差值才随时间连续变化，即当$\Delta t \to 0$时，$\Delta f \to 0$。

有了帧差模型，就有了运动分析的工具，可以在各种运动情景下方便地对目标和背景图像进行分析，从而达到检测目标出现、检测目标轨迹和识别目标等各种目的。基于帧差的运动分析是一种全局运动分析，它是以整个图像为单位进行运动分析的。下面介绍局部的运动分析方法，也就是将图像分成许多块，以块为单位进行运动分析。

7.1.2 基于块的二维运动分析

基于块的运动分析是运动分析最通用的算法之一。其主要思想是把帧图像分成一定大小的图像块，认为每个图像块具有一个唯一的运动向量（包含运动方向和距离）。通过对图像块的运动分析，找出前后帧图像各部分的对应关系。要对块运动进行分析，首先要建立块的运动模型。然后再对块的匹配方法进行研究。

最简单的块运动模型是平移。设块的大小是$N_x \times N_y$，块 B 的中心为(x_c, y_c)，经过一帧运动到新的位置，则块 B 中所有点可表示为

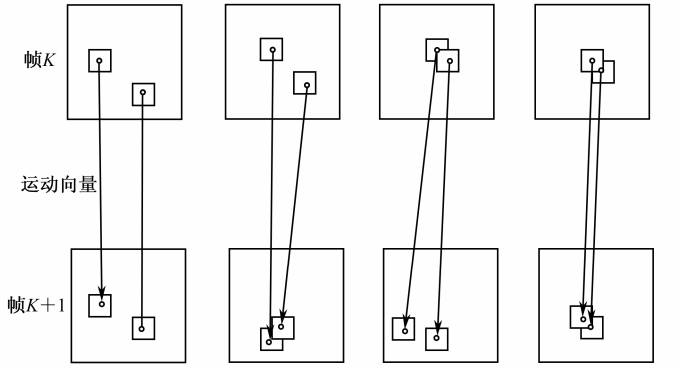
$$s(x,y,k) = B(x+\Delta x, y+\Delta y, k+1) \tag{7-9}$$

向量$(\Delta x, \Delta y)$为从(x_c, y_c)指向$(x_c+\Delta x, y_c+\Delta y)$的运动向量。一般情况下$(\Delta x, \Delta y)$取为整数，在高精度下也可以取实数。平移块运动模型如图 7.2 所示。

在图 7.2 中，整个块被认为具有单一的运动向量，可以直接在逐像素对比的基础上通过匹配来自 K+1 帧中相应块的灰度级或颜色信息得到运动补偿。图中有 4 种情况下的块平移。但是实际意义上的基于块的矢量计算，不只是考虑平移这样的简单情况，而要考虑物体变形、旋转等多方面的因素。一般采用的方法是利用可变形块进行匹配和矢量计算。这方面，已经有人利用 6 参数或 8 参数仿射变换模型来解决这个问题。如果在平移运动的同时还包

含了旋转和变形，那么就要用 6 参数仿射变换。其变换公式为

$$\begin{aligned} x' &= a_x x + b_x y + c_x; \\ y' &= a_y x + b_y y + c_y; \end{aligned} \tag{7-10}$$



（a）完全不重叠（b）运动前未重叠（c）运动前重叠（d）运动前后重叠

图 7.2　平移块运动模型

其中 a_x, b_x, a_y, b_y 称为待定参数。该变换可以处理平移、旋转及二维形变。除了仿射变换，还有透视变换和双线性变换。透视变换公式为

$$\begin{aligned} x' &= \frac{a_x x + b_x y + c_x}{a_{xy} x + b_{xy} y + 1} \\ y' &= \frac{a_y x + b_y y + c_y}{a_{xy} x + b_{xy} y + 1} \end{aligned} \tag{7-11}$$

双线性变换公式为

$$\begin{aligned} x' &= a_x x + b_x y + c_x xy + \Delta x; \\ y' &= a_y x + b_y y + c_y xy + \Delta y; \end{aligned} \tag{7-12}$$

图 7.3 是空间块平移的例子。

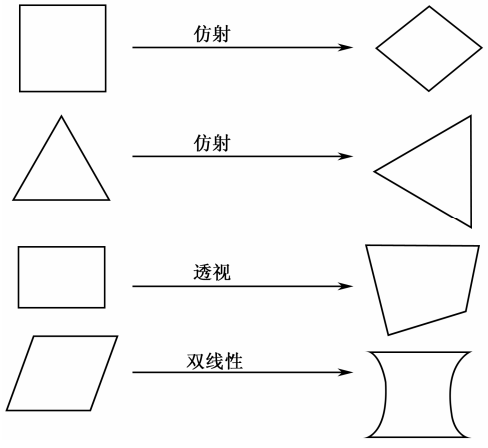


图 7.3　空间块平移的例子

如何准确快速地在 K 帧和 K+1 帧中找到对应的块是块分析的关键。其基本方法有相位相关法、块匹配法和分级运动估算法。

1．相位相关法

K 帧和 K+1 帧中两个图像块的相关函数幅值由两个块图像的相似程度决定。两个图像块越相似，相关函数幅值就越大；两个图像块相同时，相关函数幅值达到最大。K 和 K+l 帧图像块依次进行相关运算，利用相关函数幅值进行判断，就可以找到对应的相似（或相同）块。

相位相关法的优点是对亮度变化不敏感，同时可以处理多物体的运动。其缺点是，由于需要进行傅里叶变换，计算量较大；同时块的大小不好确定。为了计算较大的位移向量，块窗口必须足够大；为了保证运动向量的精度，块又必须足够小。

2．块匹配法

块匹配法运算简单，是一种最通用的算法。其基本原理如图 7.4 所示。

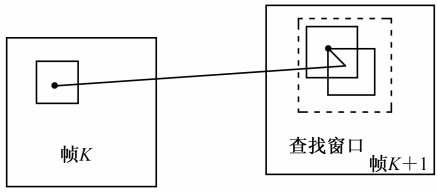


图 7.4　块匹配算法原理图

设在 K 帧中有图像块 $S(x,y)$，则在 K+1 帧中的区域 (N_x+M_x, N_y+M_y)（称为搜索窗口）中，查找与 K 帧中块 $S(x,y)$ 最匹配的图像块。在块匹配算法中，要确定匹配法则、搜索方法和块大小。

（1）匹配法则。常用的匹配法则需要根据具体的应用程序来选择。常用的匹配法则如下。

最小均方差误差函数（MSE）：

$$\mathrm{MSE}(\Delta x,\Delta y)=\frac{1}{N_xN_y}\sum_{(x,y)\in B}[s(x,y,k)-s(x+\Delta x,y+\Delta y,k+l)]^2 \tag{7-13}$$

最小平均差值函数（MAD）：

$$\mathrm{MAD}(\Delta x,\Delta y)=\frac{1}{N_xN_y}\sum_{(x,y)\in B}\left|s(x,y,k)-s(x+\Delta x,y+\Delta y,k+l)\right| \tag{7-14}$$

最大匹配像素统计（MPC），及对应位置上相近（相同）像素的个数：

$$\begin{aligned}&\mathrm{MPC}(\Delta x,\Delta y)=\sum_{(x,y)\in B}T(x,y;\Delta x,\Delta y)\\&T(x,y;\Delta x,\Delta y)=\begin{cases}1, & \left|s(x,y,k)-s(x+\Delta x,y+\Delta y,k+l)\right|\leqslant t\\0, & \text{其他}\end{cases}\end{aligned} \tag{7-15}$$

（2）匹配搜索方法。搜索方法有全搜索法、三步搜索法和交叉搜索法等。

全搜索法是搜索整个区域，$(\Delta x, \Delta y)$遍历搜索区域的每一个点。这种方法得到的结果是最优的，但也是最费时的。为此，通常使用部分点搜索策略。一类方法是先在比较稀的网格上搜索，然后在部分点作密网格搜索。另一类是采用快速搜索法和交叉搜索法。无论采用何种快速搜索法都只搜索了部分位置，搜索次数大大减少，速度大大加快，但是只能得到次优解。不过在许多情况下，次优解已可以满足要求。

（3）块尺寸的选择。块尺寸的选择非常重要。块太小，一个匹配也许被建立在包含相同灰度等级模式块上，但从运动角度来看这些模式是不相关的；块太大，在一个块中实际的运动向量存在差异，与每一块具有单一的运动向量这一假设相冲突。为此，采用分级块匹配法可以更好地解决这一问题。

3．分级块匹配法

分级块匹配法的基本思想是从最低分辨率级图像开始，在每一层依次进行运动估算。较低分辨率级用于确定相对较大块的粗略估算。接着把低分辨率级的位移向量的估算值传递到下一个高分辨率级用做初始值。较高的分辨率用于精确调整位移向量估算。图 7.5 是分级块匹配原理图。

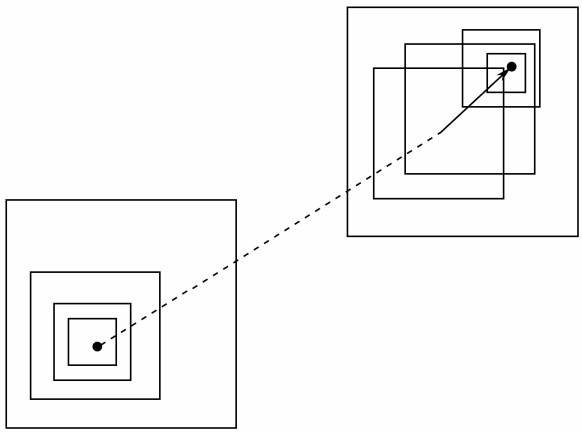


图 7.5　分级块匹配原理图

7.1.3　基于光流场的二维运动分析

在运动分析中，可以以整个图像帧为单位，通过计算帧差来直观地分析目标物体的运动。也可以以图像块为单位，通过块匹配进行物体的运动分析。而光流场的方法一般是根据当前帧及前后帧的信息，计算每个像素的运动失量。

由于视频图像是三维场景的二维投影图像。因此二维运动分析实际上是三维运动的投影分析。从三维到二维，难免丢失了许多信息，所以从根本上讲，二维分析不能得到完整的唯一解，只能在一定的假设条件下，获得场景物体运动的部分近似解。当物体运动时，

无论平移还是转动，投影平面上各对应点都有随时间和空间变化的位移和速度，分别形成二维位移场和二维速度场。它们并不都能通过图像的亮度变化来进行观测。能够通过亮度变化观测到的位移场和速度场叫做“对应场”和“光流场”。

以亮度 $s_c(p,t)$ 的变化为依据，从时刻 t 到 t' 的图像平面坐标 p 的位移称为一个对应向量，该对应向量随坐标分布形成一个对应场。在特定点 (p,t) 上的图像平面坐标的瞬时变化率，即

$$(v_x, v_y) = \left(\frac{\mathrm{d}x}{\mathrm{d}t}, \frac{\mathrm{d}y}{\mathrm{d}t}\right) \tag{7-16}$$

称为光流向量。光流向量随坐标分布就形成一个光流场。在理论上讲，当 $\Delta t = t' - t$ 趋近于 0 时，光流向量和对应向量是相同的。

总之，二维位移场和速度场是三维场景在平面上的投影。而对应场和光流场是由时变图像亮度特性得到的速度和位移函数。在一般情况下假设二维位移场和速度场等同于二维运动场。

由于二维运动有“不适定”性质，因此，运动估算算法需要有关二维运动场结构的附加假设模型，即二维运动场模型。二维运动场模型分为参数模型和非参数模型。参数模型是为了描述三维运动（位移和速度）在图像平面上的正交或透视投影。

对于参数模型，其主要缺点是它只适用于三维刚体运动。在不使用三维刚体运动模型下，也即不使用参数模型的情况下，可以将非参数均匀性（平滑度）约束条件强加于二维运动场上。非参数约束条件可归纳为确定性和随机性的平滑度模型。常用的非参数方法有下面几种。

（1）基于光流方程（OFE）的方法：依据时空图像亮度梯度来得到光流场的估计。

（2）块运动模型：假设图像是由运动的块构成的。

（3）像素递归法：依据像素上位移帧差（DFD）的梯度最小值，对预测做进一步修正。

（4）贝叶斯法：利用随机平滑约束条件。

下面着重对基于光流方程的方法进行介绍。首先建立光流方程。

设 $s_c(x,y,t)$ 表示连续的时空间亮度分布。假设亮度保持不变，则有

$$\frac{\mathrm{d}s_c(x,y,t)}{\mathrm{d}t} = 0 \tag{7-17}$$

其中，(x, y) 沿着运动轨迹随时间 t 变化。式（7-17）表示沿着运动轨迹的亮度变化率。展开上式，就得到

$$\frac{\partial s_c(p,t)}{\partial x} v_x(p,t) + \frac{\partial s_c(p,t)}{\partial y} v_y(p,t) + \frac{\partial s_c(p,t)}{\partial t} = 0 \tag{7-18}$$

其中，p 表示空间坐标点（x, y）。$v_x(p,t) = \mathrm{d}x/\mathrm{d}t$、$v_y(p,t) = \mathrm{d}y/\mathrm{d}t$ 表示依据连续空间坐标的速度向量分量。式（7-18）称为光流方程（OFE）或光流约束条件。从光流方程中可以看出，方程含有两个未知量 $v_x(p,t)$ 和 $v_y(p,t)$。一个方程要求出两个未知量，必须要给其附加一个约束条件，也称为约束方程。

基于光流方程的方法是为了求解光流方程，所以必须给其附加约束条件。根据提供的

约束条件的不同。光流方程法又分为：

（1）二阶微分法：假设空间图像的梯度在时间上守恒。不适于旋转、缩放。

（2）块运动模型（Lucas-kanade 算法）：假定在一定像素块上运动向量保持不变。它不能处理旋转运动。

（3）Horn-Schunck 法：假设光流方程中有误差项 $\varepsilon_{of}\left(v(p,t)\right)$，当 $\varepsilon_{of}\left(v(p,t)\right)$=0 时，满足光流方程 OFE。

（4）梯度估算：利用有限差分和多项式拟合两种方法来获得较稳定的偏微分估算。

以上是基于光流方程方法的基本思想。本章后面部分还会就光流算法的基于开发包实现进行具体介绍。

7.1.4　基于像素递归的二维运动分析

像素递归法是对预测值进行校正的校正器型估值器，具有如下形式

$$\hat{d}_a(p,t;\Delta t)=\hat{d}_b(p,t;\Delta t)+u(p,t;\Delta t) \tag{7-19}$$

其中，$\hat{d}_a\left(p,t;\Delta t\right)$ 代表在位置 p 处，时刻 t 时的估算的运动向量，$\hat{d}_b\left(p,t;\Delta t\right)$ 代表预测的运动向量估值，$u\left(p,t;\Delta t\right)$ 为修正项，下标 a 和 b 分别代表在像素位置（p, t）修正后与修正前的值。式（7-19）通常用于递归形式。通过在（p, t）上执行一次或多次迭代，接着进而到扫描方向上的下一个像素，因此得名“像素递归法”。

光流方程法是以光流方程作为光流限制条件的，与之类似像素递归是以位移帧差（DFD）最小值作为限制条件。DFD 的定义为：设在 t 和 t' 之间的时间段上定义 DFD 为

$$\mathrm{DFD}(p,d)=s_c(p+d(p,t;\Delta t),t+\Delta t)-s_c(p,t) \tag{7-20}$$

其中，$s_c(x,y,t)$ 代表时变图像分布，而 $d\left(p,t;\Delta t\right)\approx d(p)=\left[d_x(p)\ \ d_y(p)\right]^{\mathrm{T}}$ 代表位移向量场。可以看出，如果 $d(p)$ 等于位置 p 上的真实位移向量，在光流限制条件下，则 DFD 在那个像素得到的值为 0。通过推理可以证明：DFD 等于零的位移估算与 OFE 速度估算，在 Δt 趋于零时是等价的。

但事实上，$\mathrm{DFD}(p,d)$ 对任何 (p,d) 值几乎不曾达到绝对零值，因为有观测噪声、遮挡及插值等引入的误差。为了估算二维运动场，通常使 DFD 二次方左边达到最小值。基于梯度优化技术的像素递归法，依据预测步骤中的隐含平滑限制条件，使大幅度 DFD 二次方达到最小值。

总之，像素递归估算法的目的就是要寻找一种预测算法，使位移帧差达到最小，它实际上是一个求解全局极小值的数学问题。求解最小值的算法主要有 Newton-Raphson、Netravali-Robbins 和 Walker-Rao 方法等。基于维纳（Wiener）估算算法是 Netravali-Robbins 算法在块运动情况下的推广。所有的像素递归法均能被分级使用，利用图像的多分辨率表达式得到改进的结果，使用随机运动场模型的贝叶斯运动估算法也是由像素递归法演变而来的一种运动估算方法。

7.2 运动分割

运动目标检测是运动分析的主要研究内容之一，它是目标跟踪和目标识别的基础。在视频图像中，可以通过分析前后相邻图像帧的变化来确定运动部分的图像或者目标。对于运动目标的检测，人们提出了许多方法。其中按照摄像头与背景之间是否有相对运动，分为静止背景和运动背景情况下两种分析方法。本节主要讨论静止背景条件下运动目标的检测方法。

在静止背景下，运动目标检测的方法一般可以分为背景差分法、帧间差分法、背景模型法及基于光流的方法等。背景差分法就是用当前帧图像与已知背景图像做差来检测目标的方法。帧间差分法就是用当前帧图像与相邻帧图像进行差分来检测目标的方法。背景模型法是建立一个模型来模拟背景图像，用当前帧图像的像素点值与背景模型比较来确定是目标像素还是背景像素，从而检测到目标，可以认为它也是一种背景差分法。基于光流的方法是根据图像的光流场分布的变化来检测运动目标的方法。

上述方法理论上都能理想地检测到目标。但是在实际使用当中，由于各种外界条件的干扰，使得运动目标的检测变得比较复杂。其复杂性主要表现在以下几方面。

（1）光线变化。由于时间变化，如一天的早午晚，日光的照射强度和角度变化会引起光线变化；再由于天气原因，如晴、阴、雨、雾天及风沙等，同样会导致照射光线的变化。由于光线的变化，要使一种检测算法适应各种光照情况是比较困难的。此外，光线的突然变化，也会使目标检测增加难度。

（2）场景中运动目标的干扰。例如，大面积区域中各种交通目标的运动，车辆的突然停止与突然启动；场景中某些目标的频繁变动，如摇摆的树枝和树叶、水面的波动等。这些都会对目标的准确检测造成影响。

（3）初始化问题。在一些监视场景中，要得到无噪声干扰的纯背景图像（不含检测目标和运动背景目标的图像）是很困难的，如在人车繁忙的交通场景下。

（4）遮挡与孔洞问题。要检测的运动目标被背景中的目标所遮挡的情况下，怎样判断遮挡；颜色均匀的运动目标，有可能检测不到其内部的像素。

（5）阴影问题。在目标检测当中，如何将被检测目标与其产生的阴影区分开来，从而仅检出目标部分。

（6）目标的失踪。运动目标长期停留在场景中，有可能被看做背景。

由于目标检测的复杂性，想要建立一个通用的、适合所有情况的目标检测算法是不现实的。因此，根据具体情况建立符合实际条件的目标检测算法是目标检测方法的研究方向。下面首先介绍基于帧差分析的背景差分与帧间差分的目标检测方法，分析两种差分法的优劣，并介绍 3 种背景维护方法和一种改进的帧间差分法；然后介绍基于背景模型的目标检测方法及背景模型的建立过程；接下来介绍基于光流场的目标检测方法；最后介绍一种复杂变化背景下的运动目标检测方法。

7.2.1　基于背景差分的方法

该方法通过将当前帧与背景图像相减来检测出目标。

背景差分方法原理：首先设 B_k 为背景图象，f_k 为当前帧图像，D_k 为差分图像，则有 $D_k(x,y)=\left|f_k(x,y)-B_k(x,y)\right|$，设 R_k 为差分后二值化图像。对 R_k 进行连通性分析后，如果某一连通的区域的面积大于给定的阈值 T，则认为检测到目标出现，并认为这个连通的区域就是检测到的目标图像。一般情况下，需要首先对 R_k 进行一系列形态滤波，以克服各种噪声和干扰的影响，提高检测的准确率。其工作原理图如图 7.6 所示，实验结果如图 7.7 所示。

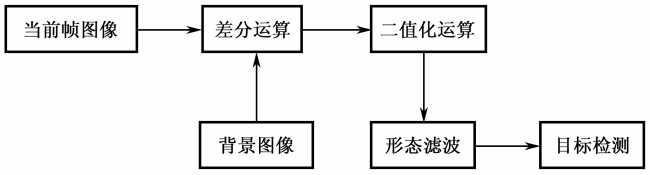


图 7.6　背景差分法工作原理图

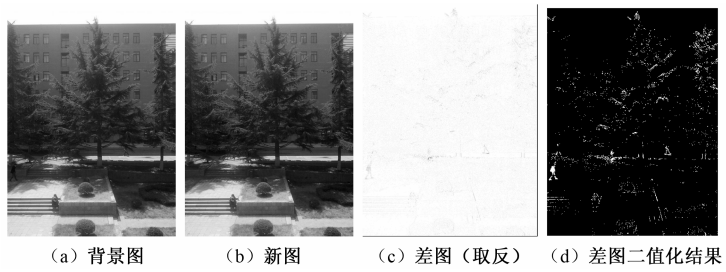


图 7.7　背景差分实验结果

背景差分方法是用当前环境背景估计图像与当前帧图像相减来实现目标检测。从理论上讲，这种方法更容易检测出运动目标。因为在理想的情况下没有噪声污染，所以差分结果一定是除了目标以外其余背景区像素差分值都为 0（全黑），目标区像素值都为 1，所以很容易提取出完整的目标图像。但是在实际应用中，要得到一幅理想的背景图像是非常不容易的。一般获取背景图像的理想方法是在场景中无任何目标时采集一幅背景图像存储起来。但是由于外界因素的干扰，如日光照射角度变化、沙尘风雨天气及目标运动频繁等因素，都会引起背景图像的不准确。为此，就需要根据外界环境变化不断地更新背景。此外，前景图像和背景图像在灰度部分上也会存在相似，导致差图像的空洞，也是需要考虑的问题。

背景差分方法的主要问题是背景的更新与维护，好的背景图象更新方法是背景差分法的关键，下面介绍相关的算法。

7.2.2 背景图像更新

由上节可知，背景差分法检测目标，方法简单、速度快、准确性高，而且能够得到完整的目标信息。但是，背景差分法的优势是建立在稳定的背景图像基础上的。用理想方法获取背景图像，在实际应用中是不实际的。例如，在交通监控等系统中，视频图像是露天环境中获取的，受昼夜光线变化、天气变化、道路场景变化、景物变化等因素的影响。因此，背景差分方法的主要问题是背景的更新与维护，好的背景图像更新方法将大大提高背景差分法的性能。下面介绍 3 种背景图像更新方法。

1．基于统计平均的背景图像更新

统计平均法就是采集多幅背景图像，然后取其平均值作为背景图像的方法。这种方法适合于场景中的目标有短暂停留行为，而且目标出现并不频繁的情况下，可以实现自适应背景更新。该算法可以用下述公式表示

$$B_k = \frac{l}{N}(f_k + f_{k-1} + \cdots + f_{k-N-1}) = B_{k-1} + \frac{1}{N}(f_k - f_{k-N}) \tag{7-21}$$

由公式可知，利用统计平均法对背景图像进行修正，其中最为关键的参数是 N。如果背景中目标出现不频繁，那么通过选择适当的参数 N，就可以获得一个较为真实的背景图像。

2．基于 IIR 滤波的背景图像更新

无限脉冲响应（IIR）滤波器是一种基于运动检测的滤波方法，一般应用于运动自适应滤波，因此我们可以考虑采用类似于 IIR 滤波器的方法来实现背景更新。可以用如下计算公式表示背景

$$B_k = (1-\alpha)\cdot B_{k-1} + \alpha \cdot f_k \tag{7-22}$$

其中α 是一个参数。可以看出，当α 较小时，可以缓慢地修正背景图像，而当α 较大时，则能够较快地更新背景图像，由此可以看出，当α 较大，而场景中有目标出现时，就会在一定程度上将目标图像叠加到背景图像上，这对目标检测来说非常不利。因此，当要用当前帧图像对背景图像进行修正时，有人就提出区别对待运动中的目标像素与背景像素。当一个像素被判定为目标像素时，则不用该像素的值对背景图像进行修正，反之就要利用该像素的值对背景进行修正。

3．基于像素分析的背景图像更新

在使用 IIR 滤波器方法对背景图像进行更新时，如果不分析新进入像素的类别，也即不区分背景像素或前景（目标）像素，则在一定程度上会将目标图像叠加到背景图像上，为解决这个问题，提出了一种分析像素类型的背景修正方法。修正公式如下：

$$B_k(x,y) = \begin{cases} \alpha \cdot B_{k-1}(x,y) + (1-\alpha)\cdot f_k(x,y), & (x,y)\text{为变化中的图像} \\ B_{k-1}(x,y), & (x,y)\text{不为变化中的图像} \end{cases} \tag{7-23}$$

其中，α 为一常数。当某一像素为前景像素时，式（7-23）会将部分前景目标的图像叠加到背景图像中，这样做的目的是为了使在场景中长期停留的目标成为背景图像的一部分。通过以上改进，利用改进方法进行背景图像更新的背景差方法检测目标的算法描述如下：

首先对第 k 帧图像中的像素类型进行区分，看其是否属于运动目标的像素，判别当前帧的像素是否属于变化像素。如果是背景像素，则不用修正，否则修正背景。判别像素类别可以利用当前相邻两帧对应像素的差值来进行。可以设定一个阈值 T，像素差值可以表示为 $|f_k(x,y)-f_{k-1}(x,y)|$，若 $|f_k(x,y)-f_{k-1}(x,y)|$ 小于 T 则认为该像素属于背景像素，否则就属于前景像素或目标像素。当完成了背景修正后，就可以利用修正后背景图像与当前帧图像进行背景差分检测和提取目标。

4．基于背景描述模型的方法

该方法通过分析场景图像背景像素值的变化特点，建立场景图像背景的描述模型，用数学模型来模拟背景图像。然后用当前帧图像中的每一个像素值与背景描述模型进行匹配，匹配上的为背景像素，匹配不上的为目标像素，然后对目标像素和背景像素进行标注，根据标注信息进行二值化处理和区域连通处理，最后检测到目标。其工作原理如图 7.8 所示。

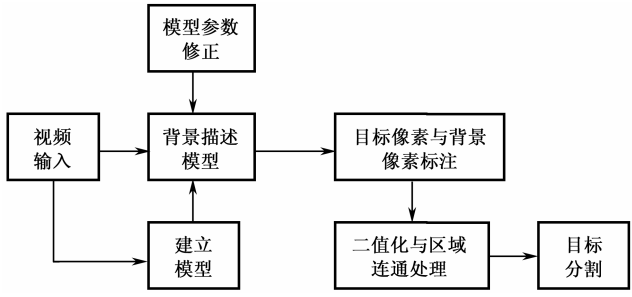


图 7.8　基于背景描述模型方法的工作原理

基于背景描述模型方法的核心是背景模型的建立与修正，下面介绍背景描述模型的产生条件和建立方法。

1）背景描述模型

背景模型有单模态和多模态两种，前者在每个背景点上的颜色分布比较集中，可以用单个概率分布模型来描述，后者的分布则比较分散，需要多个分布模型来共同描述。自然界中的许多景物和很多人造目标，如水面的波纹、摇摆的树枝、飘扬的旗帜等都呈现出多模态的特性。

单模态背景模型用单个概率分布模型来描述背景像素点的颜色分布。最常用的描述背景点颜色分布的概率模型是高斯分布。下面介绍一种基于高斯统计模型的背景模型估计方法。该方法分为两个组成部分：一是背景模型的建立，二是对背景模型的更新。

① 背景模型的建立。设 μ_0 为较长时间段内所有视频图像序列图上对应的每一像素的平均亮度，$\sigma^2{}_0$ 为像素亮度的方差，则初始背景模型为由 μ_0 和 $\sigma^2{}_0$ 确定的基于高斯分布背景图像 B_0。

$$B_0 = [\mu_0, \sigma_0^2] \tag{7-24}$$

其中，

$$\mu_0(x,y) = \frac{1}{T}\sum_{i=0}^{T-1} f_i(x,y) \tag{7-25}$$

$$\sigma_0^2(x,y) = \frac{1}{T}\sum_{i=0}^{T-1} [f_i(x,y) - \mu_0(x,y)]^2 \tag{7-26}$$

这样就完成了背景模型的建立和初始化。

② 背景模型的更新。此后需要不断对背景模型进行更新，以适应环境的变化，保证背景图像的准确性。对背景模型的更新是通过对每一幅新图像的计算，不断地自适应地更新背景参数来完成的。参数更新公式如下

$$B_t = [\mu_t, \sigma_t^2] \tag{7-27}$$

式中，B_t 为更新的背景模型。

$$\mu_t = (1-\alpha)\cdot\mu_{i-1} + \alpha\cdot f_t \tag{7-28}$$

$$\sigma_i^2 = (1-\alpha)\cdot\sigma_{i-1}^2 + \alpha\cdot(f_i - \mu_i)^2 \tag{7-29}$$

$$\alpha = K\cdot\frac{1}{\sqrt{2\pi\sigma}}\exp\left\{-\frac{(\mu_{i-1} - f_i)^2}{2}\right\} \tag{7-30}$$

α 为一个给定的[0, 1]之间的常数。

2）多模态背景模型算法

在场景中一般像素颜色值显示出非单峰分布的特点，因此单模态模型经常不能准确描述背景的变化。为此可以利用多个单模态模型组成的集合来描述场景中每一个像素值的变化情况，即用多模态模型来建立背景模型的方法。

对每个像素定义其分布模型，设像素值序列为 $\{x_{t-k}, x_{t-k+1}, \cdots, x_t\}$，在其基础上定义多个单模态模型组成的集合。

$$P_t(x,y) = \{p_{i,t} | i = 1, \cdots, K\} \tag{7-31}$$

式中，$P_{i,t} = [w_{i,b}, m_{i,t}, l_{i,t}]$、$\sum_{i=1}^{K} w_{i,t} = 1$。$P_{i,t}$ 为每一个单模态模型，它由 3 个参数组成，$w_{i,t}$ 为这个单模态模型的权值，其大小体现了当前用这个模型表示像素值的可靠程度，$m_{i,t}$ 为这个单模态模型的均值，体现了每个单峰分布的中心，$l_{i,t}$ 为这个单模态模型的单峰分布的宽度大小，其大小体现了像素值的不稳定的程度，它的作用与前述的单模态模型的作用相当。K 为单模态模型的个数，体现了像素值多峰分布的峰的个数。K 的选取依赖于像素值分布情况，同时也取决于系统的计算能力，通常的取值为 3～5 之间。不难看出，上述定义把一个像素值的观测用多个单模态模型来表示。为了使模型能不断贴近当前像素值的分布规律，需要用每一个新到的像素值更新这个模型的参数。

对于像素模型参数的修正，由于模型由多个单模态模型组成，因此它的修正与单模态模型有所不同，其参数修正步骤如下。

（1）对每个新的像素值，首先检查它是否匹配于这个模型，检测的方法如下：

$For\, i = 1 \quad K$

$if\ |x_{i,t} - m_{i,t}| < a, l_{i,t}$则匹配;

否则就不匹配

（2）针对不同情况做不同的修正。

第一种情况：新像素值与像素多模态模型集合中的某一个或几个单模态模型匹配。这时需要对单模态模型的参数进行修正，修正方法如下。

① 修正与新到像素值匹配的单模态模型的权值。

$$w_{i,t} = w_{i,i-1} + \beta \cdot w_{i,t-1} \tag{7-32}$$

式（7-32）保证模型始终能真实模拟背景像素值的最近时刻的分布情况。根据模型的定义，权重值体现了最近像素值出现的概率大小。当一个新的像素值与这个分布中某一个或几个单模态模型相匹配时，说明单模态模型较符合当前像素值的分布。这时我们需要适当增加其权重值。β 的大小体现了权重值的修正量，较大的β 值实现了较快的修正，每次的修正量为 $\beta \cdot w_{i,t-1}$。

② 修正与新到像素值匹配的单模态模型的参数 $m_{i,t}$ 及 $l_{i,t}$。

当某一单模态模型与新到的像素值匹配时，需要修正其 $m_{i,t}$ 及 $l_{i,t}$。这是因当新到的像素值与某个单模态模型匹配时，根据概率分布的观点，其势必会影响原先估计的概率分布。当单模态模型的概率分布给定时，可以用过去的观测值及新到的像素值，使用极大似然估计法重新估计模型的参数。下面以正态分布为例，说明修正方法。

当单模态模型概率密度分布为正态分布时，其最大似然分布的解可以表示为

$$m_{i,j} = \frac{1}{k+1}\sum_{j=0}^{k} x_{i-j} \tag{7-33}$$

$$l_{i,j} = \frac{1}{k}\sum_{j=0}^{k}\left(x_{i-j} - m_{j,t}\right)^2 \tag{7-34}$$

由于对参数 $m_{i,t}$ 及 $l_{i,t}$ 的重新修正，需要保存原先的观测值，这给系统设计带来了一定的困难。所以，这里的修正公式仍然使用修正单模态模型的方法去代替。

$$m_{i,t} = (1-\alpha)\cdot m_{i,t-1} + \alpha x_i \tag{7-35}$$

$$l_{i,t}^2 = (1-\alpha) l_{i,t-1}^2 + \alpha \cdot (x_t - m_{i,t})^2 \tag{7-36}$$

$$\alpha = C\frac{1}{\sqrt{2\pi l_{i,t-1}}}\exp\left\{-\frac{(m_{i,t-1}-x)^2}{2}\right\} \tag{7-37}$$

③ 未与新到的像素值匹配的单模态模型参数的修正。

当新到的像素值未与某个单模态模型匹配时，可以认为这个新到的像素值对这个单模态模型的分布没有任何作用。因此并不需要修正这个单模态模型的参数，而只是按照下面公式改变其权重就可以了。

$$w_{i,t} = w_{i,t-1} - \beta \cdot w_{i,t-1} \tag{7-38}$$

第二种情况：新到像素值未与该像素多模态模型集合中任何一个单模态模型匹配。当没有一个单模态模型与新到的像素值匹配时，说明出现了新的分布形式，而这个分布形式应在多模态模型集合中。所以此时需增加一个新的单模态模型，同时从原先的模型集合中去除一个当前多模态模型集合中权重最小的单模态模型。新增加的一个单模态模型的参数为 $w_{i,t}=\overline{W}_{i,\min};m_{i,t}=x_t;l_{i,t}=\overline{L}_{i,\max}$，其中 $w_{i,t}$ 为当前多模态模型集合中最小权重，x_t 为新到的像素值，$\overline{L}_{i,\max}$ 为一个给定的较大常数。

（3）完成上述修正后，需要对模型中各个单模态模型的权重归一化处理。

$$w_{i,t}=\frac{w_{i,t}}{\sum_{j=1}^{K}w_{j,t}} \qquad \text{其中 } i=1,\ 2\cdots K \tag{7-39}$$

3）背景像素建模

使用上述模型实现了对像素值的模拟，即对每一个新到的像素值需要判断其是否为目标像素或背景像素的模型应具有这样的特点：这个单模态模型其权重较大而且其方差较小。同时考虑这两方面因素，但很难说两者中哪一个参数对判断单模态模型是背景模型更重要。好在可以不必单独讨论这两个参数对模型是否是背景模型的影响，只需在模型集合中相互对比，以便求出那些属于背景的模型来。可以用相对值 $w_{i,t}/l_{i,t}$ 的大小作为评判标准。当然也可以对使用 $w_{i,t}/l_{i,t}$ 作为评测标准是否准确表示怀疑，但考虑到所讨论的是相对大小，所以 $w_{i,t}/l_{i,t}$ 就已经足够了。求取背景像素的模型方法如下：

（1）计算每个单模态模型的相对值 $w_{i,t}/l_{i,t}$。

（2）根据相对值 $w_{i,t}/l_{i,t}$，对每个模型按由大到小的顺序排序。

（3）取前 N 个模型作为背景模型。

7.2.3 帧间差分方法

帧间差分的基本原理是利用视频序列中连续两帧或几帧图像的差分来进行目标检测的。这与帧差的运动分析有相似之处，但又存在明显的区别。如图 7.9 所示为帧间差分法基本原理流程图。

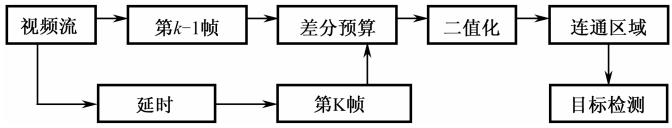


图 7.9　帧间差分法基本原理流程图

帧间差分法的工作原理主要是利用视频序列中连续的两帧或几帧图像的差异来进行目标检测。首先，计算第 k 帧与第 k−1 帧图像之间的差值，得到差值图像 D_k；然后，求取一个阈值，对差值图像 D_k 进行二值化。当差值图像中的像素值大于给定的阈值时，则认为该像素为目标像素，反之认为是背景像素。对差值图像 D_k 进行二值化后得到区域图像，对区域图像 R_k 进行连通处理，当某一连通区域的面积大于给定的阈值时，则认为检测到目标，

并认为该区域就为目标所占区域。

$$D_k(x,y)=\left|f_k(x,y)-f_{k-1}(x,y)\right| \tag{7-40}$$

$$R_k(x,y)=\begin{cases}0, & \text{背景}\quad D_k(x,y)\leqslant T\\ 1, & \text{前景}\quad D_k(x,y)>T\end{cases} \tag{7-41}$$

基于帧间差分进行运动目标检测的主要优点在于：

（1）算法原理简单，易于编程实现。

（2）由于算法中主要涉及加减运算，因此处理速度比较快，实时性好。

（3）由于采用邻近帧图像进行差分，而一般邻近帧间的时间间隔很短，因此帧间差方法对场景光线的变化一般不太敏感，抗干扰能力要优于背景差分方法。

最基本的帧间差分法可以检测到场景中的变化，并检测出目标。但在实际应用中，检测到的目标结果经常不令人满意，主要是目标的完整性要求不能很好的体现，这样就为以后的目标跟踪和目标分类造成影响。尽管帧间差分法往往不能提取完整的目标图像，但作为一种快速检测运动目标出现及运动目标的初步定位方法，常被应用。

由于存在着噪声的干扰，上面介绍的方法得到的二值化图像中往往会含有许多孤立点及孤立的小区域、小间隙和孔洞。这些都给判断背景是否存在着变化带来了干扰。因此需要将孤立的点、小区域去除，而将小间隙连接，同时又应该将小孔洞填充。这里常常采用图像形态学中对二值图像的膨胀和腐蚀等方法来完成上述处理。其中对二值图像的腐蚀和膨胀运算采用简单对称的结构元素，如 3×3 的方形结构。前面章节中已经给出了腐蚀和膨胀运算的定义。

7.2.4　目标检测

经形态滤波后，一些小干扰区域被去除，小间隙被连通，小孔洞被填充。但是相对较大的白色小区域和较大的孔洞黑色区域却仍然存在。为消除这些区域，采用阈值法来进行处理。对于那些较大的孔洞黑色区域，先设定一个阈值，然后计算各个连通黑色区域像素个数，若某一黑色区域的像素数目小于某一阈值，则将该区域改为白色区域。这样做的目的是要消除白色目标区域中由于前后两帧图像中运动目标自身相减而造成的黑洞。因为，在两帧图像中运动目标的位置往往有部分重叠，而重叠部分是造成黑色孔洞的主要因素。较大的黑色孔洞经常会干扰目标区域的检测，因为判别是否为目标区域，是根据连通的白色区域的像素数目多少来进行的，因此，消除一定大小范围内的孔洞是十分必要的。完成上述处理后，就可以通过计算各个连通白色区域的像素数目来对目标进行检测。设定一个阈值，当白色区域的像素数目大于阈值时就认为检测到目标。

7.2.5　基于光流的方法

光流是空间运动物体被观测面上的像素点运动产生的瞬时速度场。其中二维速度场是三维速度矢量在成像平面上的投影。它包含了物体 3D 表面结构和动态行为的重要信息。光流法检测运动目标的基本原理是：为图像中的每一个像素点赋予一个速度矢量，这就形成

了一个图像运动场，在运动的一个特定时刻，图像上的点与三维物体上的点一一对应，这种对应关系可由投影关系得到，根据各个像素点的速度矢量特征，可以对图像进行动态分析。如果图像中没有运动目标，则光流矢量在整个图像区域是连续变化的。当图像中有运动目标时，目标和图像背景存在相对运动，运动目标所形成的速度矢量必然和邻域背景速度矢量不同，从而检测出运动目标及位置。采用光流法进行运动目标检测的问题主要在于大多数光流法计算耗时，实时性和实用性都较差。但是光流法的优点在于光流不仅携带了运动目标的运动信息，而且还携带了有关景物三维结构的丰富信息，它能够在不知道场景的任何信息的情况下，检侧出运动对象。

对许多对实时性和准确性要求较高的系统来说，纯粹使用光流法来检测目标不太实际。更多的是利用光流计算方法与其他方法相结合来实现目标检测和运动估计，如差值图像与光流法相结合的运动目标检测和估计方法。该方法的主要思想是，先利用帧间差分法获得差值图像，然后再计算差值图像中不为零处的光流，通过分析光流场分布的变化来检测运动目标。这种方法不仅可以使计算出的光流场分布更为可靠和精确，而且同时减少了计算量（因为只是计算局部光流）。

图像差分的目的是找出差值图像中不为零处的图像的像素，也就是图像发生变化的像素，以便计算这些像素的光流分布。其原理描述如下：

设 $f(x, y, t_i)$和 $f(x, y, t_j)$分别代表时刻 t_i 和 t_j 的图像，可以得到其差值图像不为零的像素。

$$\Delta f(x,y)=\begin{cases}1, & \left|f(x,y,t_i)-f(x,y,t_j)\right|>T\\ 0, & \text{其他}\end{cases} \tag{7-42}$$

式中，T 为阈值。差值图像中$\Delta f(x,y)$为 0 的像素对应前后两时刻间没有发生变化的地方；$\Delta f(x,y)$为 1 的像素代表两帧间发生变化的地方。而发生的变化恰恰是由于目标发生运动所引起的。

除了采用两帧图像作差以外，还可以采用连续 3 帧图像来检测运动目标，这时判断一个像素点是否在运动目标上的原则如下：

$$\Delta f(x,y)=\begin{cases}1, & \left|f(x,y,t)-f(x,y,t-1)\right|>T\text{且}\left|f(x,y,t-1)-f(x,y,t)\right|>T\\ 0, & \text{其他}\end{cases} \tag{7-43}$$

式中，T 为一个经验阈值。本算法采用连续两帧图像的差分方法。

基本光流方程和计算已经在前面介绍过了。在实际应用中，光流场基本方程的速度不变条件往往难于满足，如遮挡、多光源、透明性等原因。只有当灰度的梯度很大时，基本方程才成立。用帧间差得到的差值图像中不为零的像素点，常常对应于灰度梯度较大的点。因此用帧间差和光流法相结合的方法，就能使计算出的光流场分布更为准确和有效。为准确求出光流场分布，人们在光流场基本方程的基础上，施加了很多约束条件，从而形成许多计算方法，如微分法、匹配法、频域法和马尔可夫随机场方法等。在实际应用中，基于微分的最小加权二乘法可取得较好的效果，该算法由 Lucas 和 Kanade 提出，其思想是在像素(x,y)的邻域内，用加权最小二乘法来求该点的光流，该方法的约束条件可以用下式描述：

$$\min\sum_{x\leqslant\Omega}\boldsymbol{W}^2(x)[\nabla p(x,y,t)\cdot V+\frac{\partial p}{\partial t}]^2 \tag{7-44}$$

对于在邻域 Ω 内的 n 个点，其解可以表示为

$$\boldsymbol{V}=[\boldsymbol{A}^{\mathrm{T}}\boldsymbol{W}^{2}\boldsymbol{A}]^{-1}\boldsymbol{A}\boldsymbol{W}^{2}\boldsymbol{b} \tag{7-45}$$

其中

$$\boldsymbol{A}=[\nabla p(x_1,y_1),\cdots,\nabla p(x_n,y_n)]^{\mathrm{T}}$$

$$\boldsymbol{W}=diag[W(x_1,y_1),\cdots,W(x_n,y_n)]$$

$$\boldsymbol{b}=-\left(\frac{\partial p(x_1,y_1)}{\partial t},\cdots,\frac{\partial p(x_n,y_n)}{\partial t}\right)^{\mathrm{T}}$$

∂p 表示图像灰度 p 的空间梯度。矩阵 $\boldsymbol{A}^{\mathrm{T}}\boldsymbol{W}^{2}\boldsymbol{A}$ 的特征值决定光流的可靠性。设其特征值为 λ_1 和 λ_2，如果它们均大于给定的阈值 λ，则计算出的光流是可靠的，否则是不可靠的。

实际情况中，由于光流场的不连续性，以及其违反守恒假设条件的光流场分布比计算整个运动物体的光流场要可靠得多。这时，因为它们往往对应于灰度梯度较大的点，而这些点的光流场基本方程近似成立。采用该方法，可使得计算出的光流分布更为可靠和精确，同时也减少了计算量（因为只计算了差值图像中不为零处的光流场分布，而没有计算整帧图像），提高了速度。

算法实现按以下的步骤进行：

（1）采用中值滤波先对图像进行预处理。

（2）使用帧间差法求出相邻两帧的差值图像。

（3）求出差值图像中不为零处的像素。

（4）求出不为零处像素的平均光流场分布。

（5）分析光流分布，检测运动目标。

7.3　运动目标跟踪

基于动态图像序列的运动目标跟踪技术在军事、国防、工业过程控制、医学研究、飞机导航，以及智能交通领域有着广泛的应用。运动目标跟踪就是在一段序列图像中找出其中感兴趣的运动目标在连续多帧图像中的位置序列。运动目标跟踪的目的就是通过对序列图像进行分析研究，计算出运动目标在连续帧图像中的位移、运动速度、运动目标的数量及运动目标的运动轨迹等运动参数，为图像的高层理解和分析提供依据。由于含有运动目标的序列图像比静止目标的一帧图像提供了更多的有用信息，所以从动态序列图像中可以获得在单帧静态图像下很难得到的目标信息。目标跟踪的原理就是在下一帧图像中找出目标确切位置。一般的跟踪方法是首先提取被跟踪目标的图像，建立一个模板，然后在下一帧图像中进行全图匹配，搜索目标图像，直到找到匹配的位置。

目标跟踪的详细过程描述如下：

（1）在视频序列图像中检测新的运动目标和运动区域。

（2）对新检测的运动目标进行分割。

（3）提取新运动目标的特征，并建立目标匹配模板。

（4）用预测模型预测目标在下一时刻可能出现的位置，确定下一时刻目标的搜索范围。

（5）用前一时刻的目标模板在预测的搜索范围进行匹配搜索，寻找最佳匹配位置，如果在预测范围内未找到目标时，就要进行特殊情况处理（如遮挡情况、暂停情况等）。

（6）利用匹配到的目标图像，修正被跟踪目标的模板数据，这个过程不断的交替重复。

如图 7.10 所示为目标跟踪流程图。

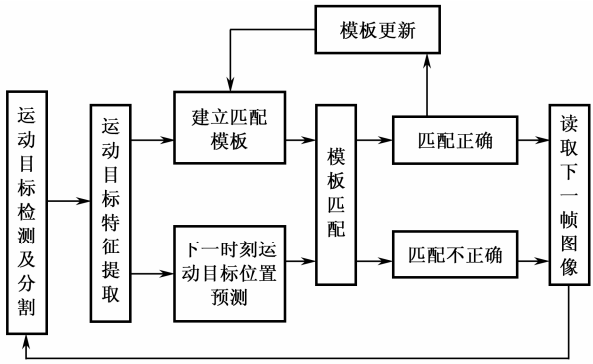


图 7.10　目标跟踪流程图

在实际中，对运动目标的跟踪并不像描述的那样简单，存在一定的难度。运动目标跟踪的主要困难是由于实际环境中目标运动的复杂性及视频数据所具有的特殊性造成的。目标在图像中的大小、运动状态、摄像机的运动与否、目标的运动轨迹、前景目标与背景的对比程度及背景的稳定程度等因素，均使得在视频图像中对运动目标进行跟踪变得复杂而困难。用视频监视技术进行运动目标跟踪的主要难度在于以下几点。

1．外界因素的干扰

由于所处的实际环境不同，运动目标的跟踪会受到不同因素的影响，这些因素会不同程度地影响运动目标跟踪的准确性和稳定性。这些因素主要体现在：

（1）光线的变化。由于场景中光线的改变，使背景图像也随之发生变化，从而很难将这些变化与由于前景目标的进入和运动造成的前景变化区分开来。

（2）背景中景物的变化。当在背景中增加或移去某些背景目标时，或者当背景中景物的位置发生变化时，只要这些变化持续一段时间，背景就需要及时用背景模型更新。

（3）目标阴影与遮挡。运动的前景目标的阴影部分可能会引起背景中局部画面亮度的变化。此外，运动的目标之间、运动目标与背景目标之间可能会发生相互重叠与遮挡，这些都会对目标特征的确定及跟踪过程中的特征匹配带来困难。

（4）前景目标在灰度、颜色或形状上的相似，使从背景中分辨出跟踪目标变得比较困难。

（5）非静态背景的影响。背景并不完全静止，如风中的树叶、波动的水面等，在这种情况下，运动背景有可能被当作前景目标进行处理，增加了对前景目标跟踪的难度。

（6）运动目标的高速运动。前景目标的高速运动可能会导致许多不同的目标频繁在背

景中出入，从而难以分辨哪些是真正的背景、哪些是要跟踪的前景目标，给目标跟踪增加了难度。

2．跟踪特征难于选择

运动目标特征的选择对于跟踪的准确性来说至关重要。与静态图像不同，运动图像是一类特殊的图像序列，在运动图像中含有大量的运动信息，时空图像中的前景目标的梯度信息、深度信息和光流场，彩色图像中的颜色、纹理特征、图像的直方图及运动序列图像间的时间相关等信息。这些信息都可以作为跟踪目标的特征。但是，我们研究的运动目标，都是目标的三维运动到二维平面上的投影，也就是说，是三维信息在二维平面上映射为二维信息，在这个映射过程中不仅目标外形可能会改变，而且有些特征信息会丢失。我们所获得的二维特征信息要想很好的反映三维目标，本身就有很多问题。因此，如何选取和提取合适的跟踪特征以解决目标三维运动给目标匹配带来的困难是一个难点。

3．跟踪的实时性要求

视频监视技术中，实时性要求是十分重要的，如果失去实时性，那么视频监视的作用就会大打折扣。作为视频监视技术中最关键技术之一的运动目标跟踪技术更是要强调实时性。由于视频数据量巨大，一般来说，图像的数据量大约为 1MB，如果按 30 帧/秒的播放速度，则每秒钟的数据量就相当庞大，如果不选择合适的算法策略，很难达到实时性要求。此外，在满足实时性的条件下，还要尽可能保证准确性。但是跟踪的准确性来自于较复杂的算法和烦琐的运算。因此，跟踪算法的又一难点在于采用何种方法策略提高跟踪算法的运算速度和匹配准确性。下面，我们介绍几种常用的运动目标跟踪方法，分析比较各方法的优劣。

7.3.1　基于特征的跟踪方法

用于目标跟踪的目标特征多种多样，无论是非刚体运动目标还是刚体运动目标，在序列图像中相邻的两帧图像，由于图像序列间的时间采样间隔很小，可以认为这些个体特征在运动形式上具有连贯性。因此，可以使用点、直线和曲线等个体特征来跟踪目标。参考文献[61]介绍了灰度图像中一种边缘直线匹配的算法。在提取边缘直线的过程中，首先利用边缘检测和边缘增强技术，对图像进行平滑去噪，突出边缘信息，再从边缘中分割出直线，并从中提取直线，然后再用以直线的几何关系和灰度图像信息为基础的匹配函数来描述两幅图像边缘直线的相似性，在连续帧图像中采用直线匹配的方法进行运动参数的估计。

基于特征的跟踪方法的优点有以下几点。

（1）由于运动所具有的平滑性，以及使用的符号模型运动方式简单，所以目标跟踪的算法相对简单，速度快。

（2）因为这种方法已经假设特征符号与运动是相互独立的，因此在运动分析时，可以不去区分运动物体是刚体还是非刚体，不用考虑其几何形状。

（3）在跟踪过程中，符号特征容易捕捉，能够匹配到每一个特征符号。

尽管基于特征的目标跟踪方法有其众多优点，但是其也有致命的弱点[68][69]。

（1）在运动目标进行复杂运动时，如旋转、非匀加速运动，这时刚体运动目标的特征提取就会有困难。

（2）由于运动的复杂性，目标之间不可避免的会出现遮挡，从而导致运动刚体的一些特征受到遮挡的影响而无法提取与识别。因此，目标运动初始化是基于特征的跟踪算法必须解决的问题，而这些问题的解决又会使跟踪算法变得非常复杂，这也是基于特征的跟踪算法的难点所在。

（3）由于符号参数与三维目标运动参数是非线性的，因此，在使用特征进行跟踪的过程中，噪声对恢复出的三维运动参数的影响相当敏感。

7.3.2 基于变形模型的跟踪方法

变形模型分为自由变形模型和参数式模型。自由变形模型没有全局结构，只要满足某些一般的正则化约束，就可以表示任何形状的模型。参数式模型是已知几何形状的先验知识，并用少数参数来表示的模型。

自由式变形模型的代表是 1987 年由 Kass 等人提出的用来处理刚性物体或非刚性物体的 Snake 主动轮廓模型。Snake 模型是一种有效的分割和跟踪工具。有人用该模型来检测目标边缘和跟踪运动目标。Snake 是基于 Snake 能量的，分割和跟踪是通过能量最小化的原则进行的。Snake 能量 E_{snake} 由控制平滑度的轮廓内部能量 $E_{\text{int}}(v(s))$、吸引轮廓到特定的图像能量 $E_{\text{image}}(v(s))$和外部约束能量 $E_{\text{cout}}(v(s))$的组合来控制和约束。能量函数表示为

$$E_{\text{snake}} = \int_0^1 E_{\text{snake}}(v(s))\mathrm{d}s = \int_0^1 [E_{\text{int}}(v(s)) + E_{\text{image}}(v(s)) + E_{\text{cout}}(v(s))]\mathrm{d}s \qquad (7\text{-}46)$$

上述 Snake 模型依赖于图像中细微的变化，因为它的解决方案是建立在变化基础之上，但 Snake 模型存在初始化轮廓问题和对图像噪声非常敏感，不能解决快速运动的目标跟踪。为给定一个适合的初始化轮廓后来又提出了 B-Snake，目标轮廓用 B 样条来表达，使轮廓的表达更加有效，更加结构化。此外还有一种可膨胀的轮廓线，它降低了轮廓线初始化的敏感性。基于计算机视觉的跟踪就是通过序列图像对跟踪目标边界连续分割而进行的。Snake 分割过程是一种以能量最小化而成功找到能量最小的表面，从而找到目标的轮廓。

此外还有学者提出了 Snake 的跳跃模型理论来解决快速运动目标跟踪的问题。其原理是当序列图像中连续两帧图像不存在目标重叠现象时，Snake 的跳跃模型就可以用来跟踪目标。这种 Snake 的跳跃模型理论是假设在图像流的处理中能够获得运动方向的基础上的。从前面每帧图像中获得的 Snake 的节点，跳跃到目标的边界，在确定目标分割半径的基础上，位移到另一个区域，并且重新初始化。通过图像流信息的反复分割和跳跃来达到能量最小化。这里 Snake 的总能量 E_{snake} 由外部约束能量 E_{cout} 和吸引轮廓到特定的图像能量 E_{image} 组成，能量函数表示为

$$E_{\text{snake}} = \sum_{i=1}^{n} [E_{\text{cout}}(i) + E_{\text{image}}(i)] \tag{7-47}$$

跳跃模型的图像流操作过程如图 7.11 所示。

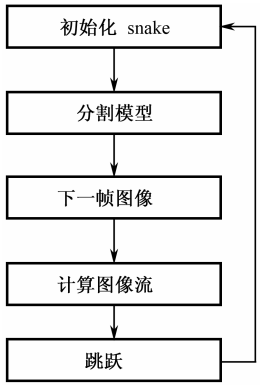


图 7.11　跳跃模型的图像流操作流程

在初始化状态中，Snake 的节点是人为初始化或自动初始化的。开始，Snake 收缩到分割目标的边界，这种分割过程是分割 Snake 总能量到最小化状态。在下一帧图像中，目标的位置将变化到另一位置，新位置与先前图像的位置很远，经过计算图像流，Snake 节点根据图像流信息进行跳跃。

Snake 跳跃模型的显著优点是可以在不连续的情况下由节点设置表示，依靠寻找最大倾斜点可以找到目标确切的边界，最大的缺点是对遮挡现象十分敏感。

7.3.3　基于区域的跟踪方法

基于区域的跟踪方法主要是依赖于前面对目标的检测来区分运动目标，然后再对目标进行跟踪。基于区域的跟踪算法依赖于两个相互影响的因素，它们是场景中投影到二维平面上的随时间变化的目标形状及其运动。运用滤波跟踪技术可以精确估计区域的几何形状及其运动速度。对该跟踪方法的一种改进是，不仅利用分割结果来为跟踪提供信息，同时也能利用跟踪信息改善分割效果。该方法的主要思想是：

（1）对当前帧图像进行帧到帧的运动分割，检测出运动区域并建立一个主控表。

（2）将主控表中的区域与检测分割出来的区域进行匹配。

（3）使用滤波器估计运动参数。

（4）用当前帧中分割出的区域来更新主控表中相应的区域。

（5）删除已移出帧图像的旧区域

（6）用主控表中每个区域相对应的运动参数去预测下一帧中运动区域的位置。

区域跟踪流程图如图 7.12 所示。

基于区域的目标跟踪方法的优点是可以同时跟踪场景中的多个运动目标，对遮挡问题不敏感，同时这种方法还可以改善图像的分割。但是，这种方法最大的问题是容易丢失目

标，跟踪的可靠性差。

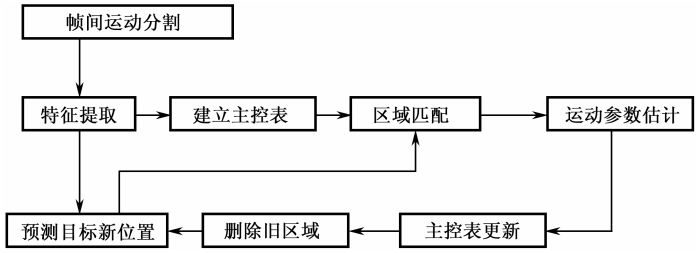


图 7.12　区域跟踪流程图

7.3.4　卡尔曼（Kalman）滤波器

卡尔曼滤波器是一种高效率的递归滤波器（自回归滤波器），它能够从一系列的不完全包含噪声的测量中，估计动态系统的状态。目前，卡尔曼滤波已经有很多不同的实现。卡尔曼最初提出的形式现在一般称为简单卡尔曼滤波器。除此以外，还有施密特扩展滤波器、信息滤波器及很多平方根滤波器的变种。最常见的卡尔曼滤波器是锁相环，它在收音机、计算机和几乎任何视频或通信设备中广泛存在。

首先引入一个离散控制过程的系统。该系统可用一个线性随机微分方程来描述

$$X(k)=\boldsymbol{A}X(k-1)+\boldsymbol{B}U(k)+W(k) \tag{7-48}$$

再加上系统的测量值

$$Z(k)=HX(k)+V(k) \tag{7-49}$$

式（7-48）和式（7-49）中，$X(k)$为 k 时刻的系统状态，$U(k)$为 k 时刻对系统的控制量。$\boldsymbol{A}$ 和 $\boldsymbol{B}$ 为系统参数，对于多模型系统，它们是矩阵。$Z(k)$为 k 时刻的测量值，$\boldsymbol{H}$ 为测量系统的参数，对于多测量系统，$\boldsymbol{H}$ 为矩阵。$W(k)$和 $V(k)$分别表示过程和测量的噪声。它们被假设成高斯白噪声，它们的协方差分别是 Q、R（这里假设它们不随系统状态变化而变化）。

如果满足上面的条件（线性随机微分系统，过程和测量都是高斯白噪声），卡尔曼滤波器是最优的信息处理器。下面结合它们的协方差来估算系统的最优化输出。

首先要利用系统的过程模型来预测下一状态的系统。假设现在的系统状态是 k，根据系统的模型，可以基于系统的上一状态而预测出现在状态

$$X(k\,|\,k-1)=\boldsymbol{A}X(k-1\,|\,k-1)+\boldsymbol{B}U(k) \tag{7-50}$$

式（7-50）中，$X(k|k-1)$为利用上一状态预测的结果，$X(k-1|k-1)$为上一状态最优的结果，$U(k)$为现在状态的控制量，如果没有控制量，它可以为 0。

到现在为止，系统结果已经更新了。可是，对应于 $X(k|k-1)$的协方差还没有更新。用 P 表示协方差

$$P(k\,|\,k-1)=\boldsymbol{A}P(k-1\,|\,k-1)\boldsymbol{A}'+Q \tag{7-51}$$

式（7-51）中，$P(k|k-1)$是 $X(k|k-1)$对应的协方差，$P(k-1|k-1)$为 $X(k-1|k-1)$对应的协方差，$\boldsymbol{A}'$ 表示 $\boldsymbol{A}$ 的转置矩阵，Q 为系统过程的协方差。式（7-50）和式（7-51）就是卡尔曼

滤波器 5 个公式当中的前两个，也就是对系统的预测。

有了现在状态的预测结果，然后再收集现在状态的测量值。结合预测值和测量值，就可以得到现在状态（k）的最优化估算值 $X(k|k)$。

$$X(k\,|\,k)=X(k\,|\,k-1)+Kg(k)(Z(k)-\boldsymbol{H}X(k\,|\,k-1)) \tag{7-52}$$

其中 Kg 为卡尔曼增益：

$$Kg(k)=P(k\,|\,k-1)\boldsymbol{H}'/(\boldsymbol{H}P(k\,|\,k-1)\boldsymbol{H}'+R) \tag{7-53}$$

到现在为止，我们已经得到了 k 状态下最优的估算值 $X(k|k)$。但是为了要令卡尔曼滤波器不断地运行下去，直到系统过程结束，我们还要更新 k 状态下 $X(k|k)$的协方差：

$$P(k\,|\,k)=(\boldsymbol{I}-Kg(k)\boldsymbol{H})P(k\,|\,k-1) \tag{7-54}$$

其中 $\boldsymbol{I}$ 为 1 的矩阵，对于单模型单测量，$\boldsymbol{I}$ 等于 1。当系统进入 k+1 状态时，$P(k|k)$就是式（7-51）的 $P(k-1|k-1)$。这样，算法就可以自回归的运算下去。卡尔曼滤波器的原理基本描述完了，式（7-50）至式（7-54）就是它的 5 个基本公式。根据这 5 个公式，可以很容易地开发实现计算机程序。

基于 Kalman 滤波器对运动目标进行估计的过程如下。

1．Kalman 滤波器参数定义及说明

设目标的运动状态参数为某一时刻目标所在的位置和速度。在跟踪过程中，由于相邻两帧图像时间间隔较短，目标运动状态变化就较小。可以假设目标在单位时间间隔内是匀速运动，所以用速度就足以反映目标的运动趋势。

根据 Kalman 滤波器的概念，可以设系统状态为 $\boldsymbol{X}_k$，$\boldsymbol{X}_k$ 是一个四维向量 (xs_k, ys_k, xv_k, yv_k)，分别表示目标在 x 轴和 y 轴上的位置和速度。而在图像上只能观测到目标的位置。定义观测状态向量为 $\boldsymbol{Y}_k$，即 (xw_k, yw_k)。

由于目标在单位时间间隔内是匀速运动的，所以定义状态转移矩阵为

$$\boldsymbol{\phi}_{k,k-1}=\begin{bmatrix}1 & 0 & \Delta t & 0\\ 0 & 1 & 0 & \Delta t\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\end{bmatrix} \tag{7-55}$$

式中，Δt 表示时刻 t_k 与时刻 t_{k-1} 的时间间隔。

由系统状态和观测状态关系可知，观测矩阵为

$$\boldsymbol{H}_k=\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\end{bmatrix} \tag{7-56}$$

此外，还可以假设 $\boldsymbol{W}_k$ 和 $\boldsymbol{Y}_k$ 都是各方向零均值且独立的噪声向量，因此设 $\boldsymbol{W}_k$ 和 $\boldsymbol{Y}_k$ 的协方差矩阵分别为

$$\boldsymbol{Q}_k=\begin{bmatrix}1.0 & 0.0 & 0.0 & 0.0\\ 0.0 & 1.0 & 0.0 & 0.0\\ 0.0 & 0.0 & 1.0 & 0.0\\ 0.0 & 0.0 & 0.0 & 1.0\end{bmatrix} \tag{7-57}$$

$$\boldsymbol{R}_k = \begin{bmatrix} 1.0 & 0.0 \\ 0.0 & 1.0 \end{bmatrix} \tag{7-58}$$

2．应用 Kalman 滤波器

在跟踪过程中，使用 Kalman 滤波器进行运动目标状态预测分为 3 个阶段，分别为滤波器初始化、状态估计和状态更新。

（1）初始化滤波器。即对 X_0 赋初值，初值为目标检测过程中得到的目标位置和速度。若速度未知，可设为零。并记录当前图像的时刻。

（2）在每帧图像进行模式搜索之前，计算与上一帧图像的时间间隔，设为Δt，代入式（7-48）和式（7-52）中，预测当前运动状态 $\boldsymbol{X}_k$，设定以 $\boldsymbol{X}^*$中 (xs_k, ys_k) 为中心的区域为搜索区域，在搜索区域中寻找模式的最佳匹配，得到 (xv_k, yv_k)，并记录下当前一帧图像的时刻。

（3）将 $\boldsymbol{Y}_k = (xw_k, yw_k)$ 代入式（7-54）更新 Kalman 滤波器状态。

7.3.5 粒子滤波器

非线性非高斯状态空间模型的最优估计在信号处理、自动控制、金融、无线通信等领域具有重要的应用。在 Bayes 框架下，最优滤波就是基于所有量测信息构造状态的后验概率分布函数（PDF）、状态的各种估计值，如均值、协方差等都可从 PDF 获得。对线性高斯状态空间模型，最优滤波就是 Kalman 滤波器，对有限状态空间是 Markov 模型（HMM），使用 HMM 滤波器或者连接树方法可得到 PDF 的解析解。然而对大量实际中出现的非线性非高斯模型，不可能得到 PDF 的解析解。

在 20 世纪 50 年代，出现了一种被称为“序贯重要性采样（SIS）”的 Monte Carlo 方法，它通过离散的随机测度逼近概率分布，并且被应用到物理和工程领域。然而由于高度的计算复杂性和退化问题，相当长一段时间内 SIS 算法没有多大进展。直到 1993 年，Gordon 提出了重采样概念，克服了早期算法的退化问题，出现了第一个可操作的 Monte Carlo 滤波器。现代计算技术使 Monte Carlo 滤波方法得到迅速发展，这些 Monte Carlo 滤波方法，在各种领域分别被称为 bootstrap、适者生存、凝聚算法、序贯 Monte Carlo 方法等，现在则通称为粒子滤波器（PF）。下面简要介绍一下粒子滤波器的原理。

动态系统可以用状态空间模型表示为

$$x_k = f_k(x_{k-1}, v_{k-1}) \tag{7-59}$$

$$y_k = h_k(x_k, w_k) \tag{7-60}$$

x_k 表示系统状态，y_k 表示量测，v_k 和 w_k 为独立同分布的系统噪声和观测噪声。假设 x_k 服从一阶 Markov 过程。给定 x_k，量测序列 y_k 相互独立，初始状态 x_0 的先验分布为 $p(x_0)$，设 $x_{0:k} = \{x_0, x_1, \cdots, x_k\}$，$y_{1:k} = \{y_1, y_2, \cdots, y_k\}$，由贝叶斯公式

$$p(x_k \mid y_{1:k-1}) = \int p(x_k \mid x_{k-1}) p(x_{k-1} \mid y_{1:k-1}) \mathrm{d}x_{k-1} \tag{7-61}$$

$$p(x_k \mid y_{1:k}) = \frac{p(y_k \mid x_k)p(x_{k-1} \mid y_{1:k-1})}{p(y_k \mid y_{1:k-1})} \tag{7-62}$$

其中 $p(y_k \mid y_{1:k-1}) = \int p(y_k \mid x_k)p(x_k \mid y_{1:k-1})dx_k$。

迭代关系式（7-61）和式（7-62）构成了最优贝叶斯解，但其解析解只对有限的模型成立，EKF、高斯混和、网格方法等逼近方法估计精度有限。利用 Mont Carlo 方法，如果能从 $p(x_{0:k} \mid y_{1:k})$ 抽取 N 个独立同分布的样本 $\{x_{0:k}^{(i)}\}$，状态的 PDF 可以用经验分布逼近为

$$\hat{p}(x_{0:k} \mid y_{1:k}) = \frac{1}{N}\sum_{i=1}^{N}\delta(x_{0:k} - x_{0:k}^{(i)}) \tag{7-63}$$

但通常不可能直接从状态的 PDF 采样，贝叶斯重要性采样方法从一个容易采样的重要性分函数 $q(x_{0:k} \mid y_{1:k-1})$ 中独立抽取 N 个样本 $\{x_{0:k}^{(i)}; i = 1, \cdots, N\}$，状态的 PDF 逼近为

$$\begin{cases} \hat{p}(x_{0:k} \mid y_{1:k}) = \sum_{i=1}^{N}\tilde{\omega}_k^{(i)}\delta(x_{0:k} - x_{0:k}^{(i)}) \\ \tilde{\omega}_k^{(i)} = \dfrac{\omega_k^{(i)}}{\sum_{i=1}^{N}\omega_k^{(i)}} \end{cases} \tag{7-64}$$

其中 $\omega_k(x_{0:k}) = \dfrac{p(y_{1:k} \mid x_{0:k})p(x_{0:k})}{q(x_{0:k} \mid y_{1:k})}$ 称为重要性权。

为了递推估计，选取重要性分布函数为

$$q(x_{0:k} \mid y_{1:k}) = q(x_{0:k-1} \mid y_{1:k-1})q(x_k \mid x_{0:k-1}, y_{1:k}) \tag{7-65}$$

从 $q(x_k \mid x_{0:k-1}^{(i)}, y_{1:k})$ 中抽取样本 $x_k^{(i)}$，重要性权 $\omega_k^{(i)} = \omega_{k-1}^{(i)}\dfrac{p(y_k \mid x_k^{(i)})p(x_k^{(i)} \mid x_{k-1}^{(i)})}{q(x_k^{(i)} \mid x_{0:k-1}^{(i)}, y_{1:k})}$，称为序贯重要性采样。为获得较好的估计，重要性分布应接近真实状态后验分布，因此重要性权的方差越小越好。在极端情况下，经过若干次迭代后，某个权可能趋于 1，其余的权都趋于 0，这称为权的退化现象，权的退化程度可以用有效样本数 N_{eff} 度量，其估计值为

$$\hat{N}_{\text{eff}} = \frac{1}{\sum_{i=1}^{N}(\omega_k^{(i)})^2} \tag{7-66}$$

为减轻权的退化，一个关键的步骤是重要性分布函数的选择，在条件 $x_{0:k-1}^{(i)}$ 和 $y_{0:k}$ 下，使权的方差最小的分布是 $q(x_k \mid x_{0:k-1}^{(i)}, y_{1:k}) = p(x_k \mid x_{k-1}^{(i)}, y_k)$，称为最优重要性分布函数。最简单的重要性分布函数是状态的先验转移分布 $p(x_k \mid x_{k-1})$，称为 Bootstrap 滤波器。

7.4　光流计算的实现

ADI 公司提供的 Blackfin 图像与视频开发包[42,43]中提供了 3 种光流计算方法的实现，分别是 Lucas-Kanade 算法、块匹配算法和金字塔光流算法。下面分别介绍其基本使用方法。

7.4.1 Lucas–Kanade 算法

库函数 API 原型如下：

```
void adi_OpticalFlowLK_i16( ADI_OPTICALFLOW *pOptical,
                            ADI_OPTICALLKBUFF *pBuffer, uint32_t nFlag );
```

该函数使用 Lucas-Kanade 方法计算 *X* 和 *Y* 方向上的光流。通过 ADI_OPTICALFLOW 结构将参数传递给该函数。需要的缓冲区要首先被初始化，并通过 ADI_OPTICALLKBUFF 传递给函数。由于 *Y* 方向光流计算的依赖性，输出会有一些延迟。ADI_OPTICALFLOW 结构定义如下：

```
typedef struct _adiOpticalFlow {
   uint8_t *pSrcA8;          uint16_t *pSrcA16;
   uint8_t *pSrcB8;          uint16_t *pSrcB16;
   uint32_t nImageStep;
   uint32_t nImageWidth;     uint32_t nImageHeight;
   uint32_t nBlockWidth;     uint32_t nBlockHeight;
   uint32_t nShiftWidth;     uint32_t nShiftHeight;
   uint32_t nMaxRangeWidth; uint32_t nMaxRangeHeight;
   float16 *pVelocX;         float16 *pVelocY;
   uint32_t nVeloStride;
} ADI_OPTICALFLOW,*ADI_OPTICALFLOW_PTR;
```

其中，pSrcA8 和 pSrcA16 分别是 8 位和 16 位输入图像 A，pSrcB8 和 pSrcB16 对应输入图像 B； nImageStep 是图像步长； nImageWidth 和 nImageHeight 是输入图像的宽和高；nBlockWidth 和 nBlockHeight 是块匹配中的块的宽和高，在 Lucas-Kanade 算法中是 Gaussian 窗口的宽和高；nShiftWidth 和 nShiftHeight 只适用于块匹配算法，分别是 *X* 和 *Y* 方向上块之间的步长；nMaxRangeWidth 和 nMaxRangeHeight 也是只适用于块匹配算法，是对给定块周围进行搜索的最大区域；pVelocX 和 pVelocY 指向 *X* 和 *Y* 方向速率缓冲区；nVeloStride 是速率步长。

ADI_OPTICALLKBUFF 结构的定义比较简单：

```
typedef struct _adiOpticalLKBuff {
   uint8_t *pBuffer;
   uint8_t *pBuffCurr;
} ADI_OPTICALLKBUFF;
```

它保存了 Lucas-Kanade 方法光流计算所需的缓冲区参数。其中，pBuffer 指向缓冲区，pBuffCurr 指向当前缓冲区位置。

函数调用前需要初始化以下变量：pSrcA16、pSrcB16、nImageStep、nImageWidth、nImageHeight、nBlockWidth、nBlockHeight、pVelocX、pVelocY 和 nVeloStride。

输入图像宽度必须是 2 的倍数。输入图像不能超过 11 位无符号整数。现在只支持 3×3

的窗口尺寸和 16 位输入。调用者需要初始化一块大小为（nImageWidth×nBlockHeight×20+ nImageWidth×8 + nBlockWidth×20）字节的缓冲区，并赋给 ADI_OPTICALLKBUFF 的 pBuffer。而 pBuffCurr 参数在第一组图像行时要置空，该指针是由算法内部使用的，因此在接下来的处理中不要去改变它。

Gaussian 计算会导致(nBlockHeight–1)/2 行的延迟。因此结果中的前 (nBlockHeight–1)/2 行是无效的。例如，对于 3×3 窗口，第一行输出是无效的。对于第 *n* 行输入，输出对应于第 *n*–1 行——如果窗口是 3×3 的。对于最后一行，输出将是第(LastLine-(nBlockHeight–1)/2) 行，因此输出中将会留下额外的(nBlockHeight–1)/2 行。这可以通过设置 nFlag 为 1 来取出。当 nFlag 是 1 时，算法基于 nImageHeight 给出余下的行。假设窗口是 5×5，要从函数中读取两个输出行，既可以设置 nImageHeight 为 2 调用该函数一次，也可以设置 nImageHeight 为 1 并调用该函数 2 次。nFlag 为 1 时不考虑图像指针，通常操作下 nFlag 应设为 0。

7.4.2　块匹配算法

开发包提供了两个块匹配算法API，分别是adi_OpticalFlowBM_i8和adi_OpticalFlowBM_i16。原型如下：

```
void adi_OpticalFlowBM_i8(
       const ADI_OPTICALFLOW * const poOpticalFlowInp,
       void *pTemp, uint32_t nUsePrev );
void adi_OpticalFlowBM_i16(
       const ADI_OPTICALFLOW * const poOpticalFlowInp,
       void *pTemp, uint32_t nUsePrev );
```

这两个函数用块匹配方法计算 *X* 和 *Y* 发的光流。两个函数分别处理 8 位和 16 位输入。参数还是使用上面介绍过的 ADI_OPTICALFLOW 结构。

在本算法中，图像被分为多个块，将以前图像帧和读取图像帧都划分为小块，然后计算这些小块的运动。算法实现中采用了螺旋式搜索方式，从前一帧中该块的原始位置开始搜索比较可能的新的位置。搜索位置如图 7.13 所示，其中 *X* 和 *Y* 方向的位移都设置为 1。每一块的光流对应块匹配中 SAD（绝对差之和）最小的位置，搜索在一个指定的范围内进行。

24	9	10	11	12		
23	8	1	2	13		
22	7	X	3	14		
21	6	5	4	15		
20	19	18	17	16		

图 7.13　螺旋式搜索方式示例

调用该函数时需要初始化以下变量：nImageStep、nImageWidth、nImageHeight、

nBlockWidth、nBlockHeight、pVelocX、pVelocY、nVeloStride、nShiftWidth、nShiftHeight、nMaxRangeWidth、nMaxRangeHeight、pSrcA16 和 pSrcB16（pSrcB8 和 pSrcB8）。还需要准备一块大小为 (oMaxRange.height+oBlockSize.hedith+1)×(oMaxRange.width + oBlockSize.width+1) ×4 字节的缓冲区给 pBuffer。

参数 pTemp 是临时缓冲区指针，nUsePrev 表明是否使用以前保存的“种子”值来找到该块的光流。

7.4.3 金字塔型光流

软件包中为金字塔型光流算法提供的库函数是 adi_TrackFeatures_i8：

```
void adi_TrackFeatures_i8(
       const ADI_PYR_OPTFL_IMAGE_FEAT_PTR poPyrOptflImgAFeatures,
       const ADI_PYR_OPTFL_IMAGE_FEAT_PTR poPyrOptflImgBFeatures,
       uint16_t nNumGoodFeatures,
       ADI_PYR_OPTFL_GD_FEAT_PTR poGoodFeatures,
       uint8_t nWindowSize,      uint8_t nMaxIteration,
       int32_t nMinDisplacement,     int16_t *pL1Buff );
```

本函数基于前一帧中通过 poGoodFeatures 返回的特征，在下一帧中跟踪那些特征。该函数处理 8 位输入，工作于两级图像金字塔结构。首先是顶级（4 下采样）处理，然后将跟踪到的位置传递给基础级，进一步提高跟踪坐标精度。在每一级上，跟踪逐次迭代，直到坐标精度小于 nMinDisplacement，或已达到最大迭代次数 nMaxIteration。对于每个特征，使用以下公式计算新的坐标位置，此处 I_x 和 I_y 分别是图像在 X 和 Y 方向的梯度，I_t 是亮度差。

$$\begin{bmatrix} \sum I_x I_x & \sum I_x I_y \\ \sum I_x I_y & \sum I_y I_y \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = -\begin{bmatrix} \sum I_x I_t \\ \sum I_y I_t \end{bmatrix} \tag{7-67}$$

参数 poPyrOptflImgAFeatures 和 poPyrOptflImgBFeatures 的以下变量需要设置：pImg（图像指针）、nImageWidth 和 nImageHeight（图像宽、高）、pSmoothImg（平滑图像指针）、pImgPyr（图像金字塔指针，按 4 下采样）、pGradXImg 和 pGradYImg（图像 X 和 Y 方向梯度的指针）、pGradXPyr 和 pGradYPyr（图像金字塔在 X 和 Y 方向梯度的指针）。还要设置 poGoodFeatures 的以下变量（对所有特征）：nCoordX（X 坐标）、nCoordY（Y 坐标）和 nVal_StatusCode（评分）。还要提供大小为(((nWindowSize×nWindowSize)+1)×6)×2 字节的 L1 缓冲区 pL1Buff。

参数 poPyrOptflImgAFeatures 和 poPyrOptflImgBFeatures 分别是图像 A 和 B 特征结构的指针，nNumGoodFeatures 是要跟踪的特征的数目。poGoodFeatures 是指向包含“好”特征坐标结构的指针，它要保存新的被跟踪坐标。nWindowSize 是要处理的窗口尺寸。nMaxIteration 是允许的最大迭代次数。nMinDisplacement 是下一次迭代要做的最小坐标位移。pL1Buffnput 是内部使用的 L1 缓冲区。

7.5　前景目标检测的实现

视频处理开发包[43]提供了一个检测前景对象的应用，演示了视频处理工具包的使用，可以用于视频监控系统中。该应用从视频文件或视频设备读入视频数据，逐帧进行对象检测，在检测到的对象周围画一个包围矩形。整个系统的软件系统结构如图 7.14 所示。

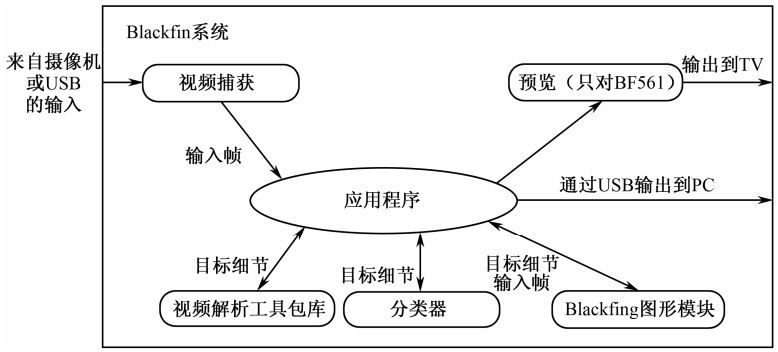


图 7.14　前景对象检测系统软件系统结构

视频解码器 ADV7183 将 ITU-R BT656 视频输入发送到 Blackfin 处理器，处理器程序检测出前景对象。EZ-KIT 使用 USB 接口将压缩过的二进制帧差图像发到 PC 主机。如果是使用视频文件输入，USB 接口还负责传输视频数据到 Blackfin 处理器。ADSP-BF561 还有一个额外的预览模式，利用视频编码器 ADV7171 输出检测到视频结果到电视上。如图 7.15 所示为演示系统的组成。

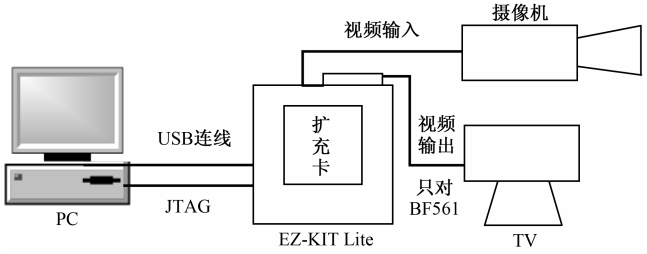


图 7.15　演示系统的组成

7.5.1　初始化对象检测库

函数 ADIOBJDETCodecNew 创建并初始化一个新的视频对象检测实例，原型如下：

```
ADICodecHandle ADIOBJDETCodecNew( ADICodecHandle hCodecHndl,
    ADIMemMap *pMemMap, ADIMemDMAService *pDMA, unsigned int *pInstID );
```

调用者必须要首先建立 ADIMemMap 和 ADIMemDMAService 对象，并分配 ADIMemMap 对象所需的内存块。主要参数如下。

hCodecHndl：是视频检测对象实例的地址，可以放到 L1 或 L2/L3 Cache 中。

pMemMap：是内存分配对象地址，可以放在 L2/L3 Cache 或 L2/L3 非 Cache 中。

pDMA：是 MDMA 对象地址。

pInstID：是保存当前实例 ID 变量的地址。应用程序如果声明多个实例，需要将该变量传到所有的初始化调用中。每个实例就会与同一个唯一的 ID 关联起来。调用成功时，返回创建的实例句柄的地址。

视频分析开发包实例结构定义如下：

```
typedef struct ADICodecInstance {
        unsigned int nSize;          ADIVersion nVersion;
ADICodecType eCodecType;     unsigned int nInstanceNum;
unsigned int nPrivDataSize;  void *pPrivData;
// 方法指针
ADICodecStatus(*pReset)(ADICodecHandle);
// 设置配置选项
ADICodecStatus(*pSetConfig)(ADICodecHandle,ADICodecConfigID,void*);
// 得到配置值（状态请求）
ADICodecStatus(*pGetConfig)(ADICodecHandle, ADICodecConfigID, void *);
// 主处理函数
ADICodecStatus (*pProcess)(ADICodecHandle, ADICodecData *,
                           void *, ADICodecData *, void *);
// 处理序列信息（如果被要求）
ADICodecStatus(*pProcessSequence)(ADICodecHandle, ADICodecData *);
// 从应用程序得到数据
ADICodecStatus(*pGetData)(ADICodecHandle, unsigned char *,
                           unsigned int *);
// 刷新编解码器内部缓冲区
ADICodecStatus (*pFlush)(ADICodecHandle, ADICodecData *, void *);
} ADICodecInstance;
```

ADIOBJDETCodecNew 创建该指针后，本实例上所有接下来的操作都通过函数指针进行调用。字段内容如下。

nSize：本结构的字节数。

nVersion：API 的版本号。

eCodecType：未使用。

nInstanceNum：该实例的实例号，未使用。

nPrivDataSize：pPrivData 数据字节数。

pPrivData：指向库私有实例数据结构。

pReset：指向重置函数，置空。

pSetConfig：设置配置参数的函数指针。

pGetConfig：获取参数的函数指针。

pProcess：在视频输入帧中进行检测的函数指针。

pProcessSequence、pGetData 和 pFlush 都置空。

对象检测实例创建后，用 pSetConfig 对 ADI_CODEC_CONFIG_OBJDET_INPUT_PARAM 对应的参数进行设置。该参数类型为 ADICodecConfigObjdetInputParam，是视频分析工具包实例的输入参数结构，定义如下。

```
typedef struct{
    unsigned int nSize;      ADIVersion nVersion;
    int ccir_width;          int ccir_height;
    int original_width;      int original_height;
    int video_standard;      int color_format;
    int enable_decimation;
    int vid_capture_width;   int vid_capture_height;
    int vid_capture_frate;   int threshold_flag;
    int output_diff;         int output_curr;
    int output_bg;           int output_compressed;
    int ccir_fld_height;     int framerate;
    int nAlphaFactor;            int numLinesInSlice;
    int pos_x;                   int pos_y;
    int maxSizeObj;          int minSizeObj;
    int nAlgorithm;
    float rVarianceThresholdFactor;     int nVarianceDeviation;
    int nLearnPeriod;    int nLearnRateEnabled;  float nLearnRate;
    unsigned int nUpdateThresholdFlag; int nUpdateTime;
} ADICodecConfigObjdetInputParam;
```

其中，nSize 是该结构的字节数；nVersion 是 API 的版本；ccir_width 和 ccir_height 是输入视频帧的宽和高；ccir_fld_height 是输入视频的场的高度。

original_width 是输入视频帧宽度，如果指定了 ROI，这就是 ROI 的宽度。例如，如果输入是 NTSC 下的 UYVY422 格式，那么它设为 720。如果输入是 CIF 格式的 YUV420 帧，就要设为 352。最小值是 64，最大值是 720。original_height 是对应的高度信息。

video_standard 仅为 color_format=ADI_ENC_CCIR422 指定，取值包括 ADI_ENC_NTSC、ADI_ENC_PAL、ADI_ENC_ACTIVE_NTSC 和 ADI_ENC_ACTIVE_PAL。

color_format 为输入视频格式，取值包括 ADI_ENC_YUV420 和 ADI_ENC_CCIR422。

enable_decimation 标识指定是否要对输入视频帧进行伸缩。

vid_capture_width 和 vid_capture_height 为输入视频捕获宽和高。不指定 ROI 时，与 original_width、original_height 相同。

threshold_flag 为像素阈值，超过阈值判定为前景，取值为 0～255。

output_diff 确定是否输出检测到的差帧图像。注意，如果设为输出差帧，则无法做到实

时处理。

output_curr 确定是否显示当前输入帧。

output_bg 确定是否输出背景帧。

output_compressed 确定是否使用 ADI 定义的压缩格式输出差帧。

Framerate 是视频输入帧速率。

nAlphaFactor 是学习背景的速率。该值越高，学习速率越慢。比如，如果值是 8，背景学习频率是 1/8=0.125 帧。最小值 1，最大值 32768，默认值 8。

numLinesInSlice 可以作为一个块处理的行数，默认值 4。

nAlgorithm 是采用的算法类型。该库支持同一个算法的 3 个版本，以提供对不同应用和场景的可伸缩性。ADI_OBJDET_ALGORITHM_BASIC 是基本算法，检测任何种类的运动，可用于检测任何定义区域的运动/入侵。这可能不支持对象的消失和离开。其 MIPS 最低。ADI_OBJDET_ALGORITHM_ADVANCED1 是同一个算法的升级版，可以在用户定义的时间段内检测入侵运动或对象的离开/消失，最适合用于室外场景。ADI_OBJDET_ALGORITHM_ADVANCED2 是基本算法的又一个升级版，与前一个功能相同，但同时适合于室内和室外场景。它需要的 MIPS 最高。注意，升级版算法只是检测离开场景的像素，使用应用程序需要进行分类等基于规则的处理以确保像素是否是离开的对象，如跟踪、基于规则的分类、模板匹配等。

nLearnPeriod 是系统的初始背景学习周期，一般设为 2～3s。

nLearnRateEnabled 确定是否要使能学习周期。如果不指定，则每帧都学习。

nLearnRate 是系统学习的速率。通常设为 0.15 或 0.1，分别对应帧速率 25 和 30。

nUpdateThresholdFlag 是从场景中消失的像素的阈值。其共有 32 位，有 3 个配置值：0～7 位是最小阈值，默认值是 6；8～15 位是最大阈值 1，默认值是 60；16～31 位是第二最大阈值，默认值是 200。

rVarianceThresholdFactor 是向差值与方差比较的阈值。该值越高，检测敏感度就越低；值越低检测敏感度越高，但是会检测到很多噪声像素。通常取值范围是[1, 2]，默认值是 1.2。

nVarianceDeviation 是标准差，取值范围是[1, 3]，默认值是 2。对于低对比度场景，可以设为 1。

nUpdateTime 是当静止的或新的对象进入场景后，需要进行背景学习的时间（帧数）。注意，在该周期后可能还有一定的延迟，延迟依赖于背景学习的速率。

pos_x 和 pos_y 是 ROI 左上角的坐标。

minSizeObj 和 maxSizeObj 是待检测目标的最小和最大尺寸。

7.5.2 基于视频开发包的前景对象检测的实现

（1）初始化。调用 VideoGrabber_Init 初始化视频“捕获”（可以是 PPI 接口，也可以是文件输入）。然后调用 VideoGrabber_Start 启动视频“捕获”。

（2）循环地处理每一帧图像。

① 调用 VideoGrabber_dequeue_frame 捕获到一帧图像。如果是 PPI 输入，则调用 adi_VideoIn_GetFrame 从视频输入队列中返回一帧捕获的图像，如果列表为空则返回 NULL。如果输入类型是文件，则通过文件读取一帧图像。

② 对本帧图像进行处理——调用 ADICodecInstance 结构中 pProcess 指向的主处理函数完成目标检测。

③ 调用 ADICodecInstance 结构中 pGetConfig 指向的函数读取类型为 ADI_CODEC_CONFIG_OBJDET_OBJECTS 对应的检测到的目标数目及相关信息。

④ 如果设置为允许，则调用 ADIClusterBlobs 对检测到的目标进行聚类。聚类过程中用到的主要数据结构是 ADICodecConfigObjdetClusterObject，定义如下：

```
typedef struct {
    unsigned int nSize;          ADIVersion nVersion;
    unsigned int nNumObjects;    int isInterMergeEnabled;
    void *pSrcObjs;              void *pDstObjs; int *pClusterArray;
    ADIOBJDETCmpFunc pfCmpFunc;
    ADIOBJDETMergeFunc pfMergeFunc;
} ADICodecConfigObjdetClusterObject;
```

该结构提供了聚类基于的准则，主要是通过设置回调函数 pfCmpFunction。库函数无需知道对象特征的任何信息。所以传递给库函数的对象类型是 void*。库函数还调用回调函数 pfCmpFunction，来决定是否将两个对象合并为一个。如果该函数返回 1，那么它将两个对象合并为一个节点。类似的，应用程序也应该通过调用回调函数 pfMergeFunction 指定是否将两个对象合并。

对于前景目标检测，聚类的基本准则是基于位置，检验两个对象是否相交、其形心是否接近、距离是否接近等。

⑤ 本帧处理结束后，调用 VideoGrabber_free_frame 释放该视频帧。对 PPI 输入，调用 adi_VideoIn_EnqueueFrame 将刚才通过调用 adi_VideoIn_GetFrame 得到的视频帧返回给视频输入驱动器。

（3）停止整个处理过程，关闭视频捕捉器。

7.5.3　前景对象检测中的基础算法

下面介绍整个算法实现中用到的一些基础图像处理函数。这些函数以动态链接库的形式给出，位于视频分析工具包基础库 libadi_vat_primitives_BF.dlb 中，对应的头文件是 adi_vat_primitives.h。

1．二值化函数 adi_Threshold_Binary_i8

```
int32_t adi_Threshold_Binary_i8( uint8_t *pInBuff, uint16_t nWidth,
```

```
        uint16_t nHeight, uint8_t nThreshold, uint8_t nLabelVal,
        uint8_t *pOutBuff );
```

该函数对灰度图形进行二值化。如果输入像素大于阈值，将用户指定的特定标记值写入输出图像，否则对应位置写 0。平均每个像素的处理需要 1.5 个机器周期。

2．绝对差函数 adi_AbsDiff_i8

```
int32_t adi_AbsDiff_i8( uint8_t *pOutBuff, uint8_t *pInBuff1,
        uint8_t *pInBuff2, int32_t nWidth, int32_t nHeight );
```

该函数计算两个 8 位图像之间的差的绝对值。输出缓冲区可以与输入缓冲区相同，也可以不同。平均每个像素的处理需要一个机器周期。

3．中值滤波函数 adi_Median_Filter_i8

```
int32_t adi_Median_Filter_i8( uint8_t *pMedBuff, uint8_t *pInBuff,
                              uint32_t nInputPixels );
```

该函数更新前一帧图像的时域均值估计。如果某个像素的像素估计值小于当前像素，就将其增加 1，否则就减 1。在一序列视频帧上，给定像素的均值估计值在任何稳态下都逼近前一帧中真正的均值。这样，估计的时域图像就可以用于在稳态下对背景的估计。该函数可以用来在一个二进制前景检测帧中删除噪声像素。平均每个像素的处理需要 2.7 个机器周期。

4．二进制中值滤波函数 adi_Median_BinaryImage_i8

```
int32_t adi_Median_BinaryImage_i8 ( uint8_t *pInBuff, uint16_t nWidth,
            uint16_t nHeight, uint8_t nFlag, uint8_t *pOutBuff );
```

该函数计算二进制图像的 3×3 均值滤波。输出缓冲区要不同于输入缓冲区。当高度为 N，应用程序必须提供(N+2–nFlag)行的输入缓冲区。例如，当函数用来处理高度为 4 的图像片段，应用程序必须提供 6 或 5 行（当 nFlag 是 0 或 1）输入图像缓冲区。应用程序必须确保第（–1）行和第(nHeight+1)行已经准备好。

nFlag 标识决定图像的最后一行是否用下一行来计算中值。如果是 1，最后一行不用下一行计算中值，只使用最后一行和前一行。如果是 0，要用最后一行计算中值，这样，应用程序必须在输入缓冲区中提供额外的行。平均每个像素的处理需要 4.55 个机器周期。

5．迭代更新背景函数 adi_Diff_RecursiveUpdatebg_i8

```
int32_t adi_Diff_RecursiveUpdatebg_i8 ( uint8_t *pInBg,
     uint8_t *pInCurr, uint8_t *pOutBg, uint16_t nThreshold,
     uint32_t nInputPixels, uint16_t nCurrMulFactor );
```

该函数基于当前视频帧更新背景帧（或称为流动均值）。它计算背景帧像素与当前帧

像素的绝对差。如果差值小于某个阈值 T，则背景像素进行如下更新

$$Bg = \left| pInBg(i) - pInCurr(i) \right| < T$$

$$pOutBg(i) = (1-\alpha)\, pInBg(i) + \alpha\, pInCurr(i), Bg = 1$$

输入缓冲区与输出缓冲区可以相同（原位计算）或不同。平均每个像素的处理需要 5.25 个机器周期。

6．前景背景分类函数 adi_ClassifyBgFg_i8

```
int32_t adi_ClassifyBgFg_i8 ( uint8_t *pInMean, uint8_t *pInCurr,
     uint8_t *pInVar, uint16_t nLabelValue, uint32_t nSize,
     uint16_t nSigma, uint8_t *pOutBuff );
```

该函数计算前景二值化帧，将每个像素分类为前景或背景。它计算当前像素值与估计均值的绝对差。如果差值大于σ 倍的方差估计值，像素分类为前景像素，否则作为背景像素。其中当前帧缓冲区会被函数覆盖，因为像素的差值会被写入该缓冲区。平均每个像素的处理需要 4 个机器周期。

其他更多函数的使用请参考 ADI 的文档“视频分析工具包基本 API 指南”。

7.6　Kalman 滤波器的实现

7.6.1　开发包中的 Kalman 滤波器 API

开发包中 Kalman 滤波器 API 主要有 3 个：adi_KalmanCreate、adi_KalmanPredict 和 adi_KalmanCorrect。

1．adi_KalmanCreate

函数原型为：

```
ADI_ITB_MODULE_STATUS adi_KalmanCreate( ADI_KALMAN *pKalman,
                    int8_t *pMatrixMemory, uint32_t nSize, uint32_t nShift );
```

这是 Kalman 滤波器的初始化函数。它初始化 Kalman 结构，并检查矩阵维数是否匹配。本函数分配并填充预测状态矩阵和协方差矩阵，并分配临时矩阵的内存，用于滤波器校正和预测阶段矩阵的临时存储。

注意，调用 KalmanCreate 结构前必须要初始化状态矩阵、测量矩阵、后验协方差矩阵和状态矩阵、状态噪声和测量噪声的协方差及控制矩阵。如果使用扩展 Kalman 滤波器，那么还要初始化测量和处理噪声的雅可比矩阵。对应的雅可比函数也要初始化。

Kalman 结构 ADI_KALMAN 定义如下：

```
typedef struct _adiKalman {
    ADI_MAT *pMatA;      ADI_MAT *pMatH;     ADI_MAT *pMatX;
    ADI_MAT *pMatP;      ADI_MAT *pMatB;     ADI_MAT *pMatQ;
```

```
    ADI_MAT *pMatR;      ADI_MAT *pMatW;     ADI_MAT *pMatV;
    ADI_MAT *pMatXPre;   ADI_MAT *pMatPPre;  ADI_MAT *pMatK;
    int8_t *pTemp;           uint32_t uKalmanGain;   uint32_t nShift;
   int8_t bIsIdentityH;      int8_t bIsIdentityA;     int8_t bIsIdentityB;
    int8_t bIsIdentityW;     int8_t bIsIdentityV;     int8_t bEnableExt;
    void (*pt2JacoAFunc) (int32_t *pState, uint32_t nStateSize, int32_t
    *pControl,
                          uint32_t nControlSize, int32_t *pJacoA,
                          uint32_t nShift);
    void (*pt2JacoHFunc) (int32_t *pState, uint32_t nStateSize,
                          uint32_t nObserSize, int32_t *pJacoH, uint32_t
                          nShift);
    void (*pt2JacoWFunc) (int32_t *pState, uint32_t nStateSize, int32_t
    *pControl,
                         uint32_t nControlSize, uint32_t nProcessNoiseSize,
                         int32_t *pJacoW, uint32_t nShift);
    void (*pt2JacoVFunc) (int32_t *pState, uint32_t nStateSize,
                         uint32_t nObserSize, uint32_t nMeasurementNoiseSize,
                         int32_t *pJacoV, uint32_t nShift);
    void (*pt2PredState) (int32_t *pState, uint32_t nStateSize, int32_t
    *pControl,
          uint32_t nControlSize, int32_t *pPredState, uint32_t nShift);
    void (*pt2PredObser) (int32_t *pState, uint32_t nStateSize,
          uint32_t nObserSize,int32_t *pPredObser, uint32_t nShift);
} ADI_KALMAN;
```

ADI_KALMAN 为当前和预测状态保存状态和协方差。字段如下：pMatA 是状态转移矩阵 $\boldsymbol{A}$，pMatH 是测量/观测矩阵 $\boldsymbol{H}$，pMatX 是后验状态向量 $\boldsymbol{X}$，pMatP 是后验协方差矩阵 $\boldsymbol{P}$，pMatB 是控制矩阵 $\boldsymbol{B}$，pMatQ 是状态过程噪声协方差 $\boldsymbol{Q}$，pMatR 是测量过程噪声协方差 $\boldsymbol{R}$，pMatW 是将 $\boldsymbol{Q}$ 映射到状态转移函数的雅可比矩阵 $\boldsymbol{W}$，pMatV 是将 $\boldsymbol{R}$ 映射到测量函数的雅可比矩阵 $\boldsymbol{V}$。

pMatXPre 是先验状态向量，pMatPPre 是先验协方差矩阵，pMatK 是 Kalman 增益矩阵 K。pTemp 是中间计算用的临时空间，uKalmanGain 是 Kalman 增益尺度因子，nShift 是定点标识。bIsIdentityH、bIsIdentityA、bIsIdentityB、bIsIdentityW、bIsIdentityV 分别标识矩阵 $\boldsymbol{H}$、$\boldsymbol{A}$、$\boldsymbol{B}$、$\boldsymbol{W}$、$\boldsymbol{V}$ 是否为单位矩阵。

pt2JacoAFunc 是返回雅可比 $\boldsymbol{A}$ 的函数指针，pt2JacoHFunc 是返回雅可比 $\boldsymbol{H}$ 的函数指针，pt2JacoWFunc 是返回雅可比 $\boldsymbol{W}$ 的函数指针，pt2JacoVFunc 是返回雅可比 $\boldsymbol{V}$ 的函数指针，pt2PredState 是返回预测状态的函数指针，pt2PredObser 是返回预测观察结果的函数指针。

其中，Kalman 结构中的以下字段必须要进行初始化：pMatH、pMatQ、pMatR、pMatA、

pMatX、pMatP、bIsIdentityH、bIsIdentityA 和 bIsIdentityB。对于扩展 Kalman 滤波器，还要初始化以下内容：pMatW、pMatV、bIsIdentityW、bIsIdentityV、pt2JacoAFunc、pt2JacobHFunc、pt2JacoWFunc、pt2JacoVFunc、pt2PredState 和 pt2PredObser。此外，应用程序还要为 pMatK 分配内存。

在以上所有矩阵中，矩阵的内存指针和矩阵的尺寸（由行、列指定）需要正确地初始化。本函数会检查矩阵维数的正确性（符合 Kalman 方程的要求）。注意，pMatXPre 和 pMatPPre（后验状态和协方差）矩阵还有 pTemp 的内存需要从 pMatrixMemory 中分配。对于普通 Kalman 滤波器，pMatrixMemory 的尺寸至少是 $N+N\times N+5\times\max(N, M)\times\max(N, M)$，对于扩展版本，至少需要 $N+N\times N+5\times\max(N, M)\times\max(N, M, Q, R)$。这里 N 是状态的数目，M 是观察结果的数目，Q 是状态噪声维数，R 是测量噪声维数。

参数中 pMatrixMemory 指向用于为临时内存、后验状态和协方差预测矩阵分配内存的缓冲区。该内存的尺寸至少是 $7\times N$，N 是 Kalman 函数状态变量的个数。nSize 是内存缓冲区的尺寸。nShift 用于定点输入数据，代表小数部分的位数。

2．adi_KalmanPredict

函数原型为：

```
ADI_ITB_MODULE_STATUS adi_KalmanPredict(
                const ADI_KALMAN *pKalman, ADI_MAT *pMatU );
```

该函数基于当前的状态和协方差数据，以及状态和测量转移矩阵，来为下一步预测状态和误差协方差矩阵。系统的预测状态可以从 pKalman 结构中的 XPreMat 和 PPreMat 矩阵结构中获取。控制矩阵在创建过程中未被初始化，需要传到预测函数——如果有预测函数的话。注意，需要先调用 adi_KalmanCreate 初始化 pKalman。

pKalman 指向 Kalman 结构，它保存本函数输出的预测状态和误差协方差矩阵。它给出的信息包括下一个位置的期望值等类似数据。pMatU 指向控制矩阵。如果没有控制矩阵，初始化为 NULL。

3．adi_KalmanCorrect

```
ADI_ITB_MODULE_STATUS adi_KalmanCorrect(
                        ADI_KALMAN *pKalman, ADI_MAT *pMatZ );
```

使用当前时间戳数据校正预测的状态和误差协方差矩阵，并更新状态和误差协方差矩阵。校正后的状态要保存到 pKalman 结构中的 pMatX 和 pMatP 结构中。预测信息的校正发生在使用观测数据的过程中，观测数据由用户通过 pMatZ 矩阵传递给本函数。它将系统的观测状态传给本函数，并基于该数据来校正预测状态的误差，以给出被跟踪系统的正确状态。

注意，需要首先调用 adi_KalmanCreate 来初始化 pKalman 结构。而且还需要调用 adi_PredictKalman 来得到预测状态。

参数 pKalman 指向 Kalman 结构，它保存正确的/更新过的状态和误差协方差矩阵作为本函数的输出。它给出关于被跟踪系统正确状态的信息。pMatZ 指向测量矩阵，在当前时

间戳上拥有观测值。

7.6.2 基于 API 的 Kalman 滤波器实现过程

在图像处理开发包中有一个 Kalman 滤波器的演示工程 kalman_filter_bf5xx.dpj。它首先读取 Kalman 输入文件，将数据转存到 L3 缓冲区；然后根据读到的信息按照上述要求填充 ADI_KALMAN 结构中的内容，并调用 ADIKalmanFilter，ADIKalmanFilter 封装了 Kalman 滤波器的功能。

（1）首先调用 adi_KalmanCreate 完成 Kalman 滤波器结构的初始化。

（2）调用 adi_KalmanPredict 进行预测，并将预测位置加 1。

（3）在限定的次数内反复迭代：调用 adi_KalmanCorrect 完成滤波器校正；如果预测位置小于最大迭代次数则调用 adi_KalmanPredict 对滤波器进行预测，同时将预测位置加 1。

迭代结束后就完成了 Kalman 滤波过程。程序最后将输出的状态向量、协方差矩阵、先验状态向量、先验协方差矩阵等写入输出文件中。

7.7 视频交通流检测系统设计

随着机动车拥有量的急剧增长，交通流量日益加大，拥塞现象日趋严重，成为一个重大的社会问题。基于视频智能处理的交通流检测及车辆识别等技术得到广泛的重视。通过对道路交通状况信息与交通目标的各种行为（如违章超速、停车、超车等）的实时检测，可以自动统计交通路段上行驶的机动车的数量，计算车辆速度，达到监测道路交通状况的作用。同时还可将检测和识别得到的交通信息存储起来，为分析和交通管理提供依据。

首先利用一组摄像头完成视频信号的采集和输入，然后利用计算机或嵌入式处理器对数字化的视频数据进行智能处理，利用计算机视觉、模式识别等数字信号处理器技术和先进的算法，获取车辆数量、车速、道路的空间占有率等重要的交通参数。这个系统应实现基本交通流参数的检测及道路实时监控功能。随着算法的逐步改进，还可以实现车牌自动定位与识别、车型识别、异常运动检测等复杂分析结果。这些功能是传统检测手段难以实现的。因此，基于智能视频处理的交通流检测技术必将占有重要的地位。

基于视频处理技术的交通流检测在国外已较为成熟，如美国的 Autosope、VTDS、ITERIS，西门子的 ARTEMIS 和比利时的 Traficon 等。美国 ITERIS 公司的 Vantage 视频交通数据采集系统采用了先进的数字视频技术，能够完美地融入道路交通数据采集系统。其视频车辆检测器具有多种交通数据采集功能：每车道交通流量、每车道平均车辆行驶速度、每车道的平均车头时距、每车道平均车辆时间间距、每车道分车型统计的车流量、车道占用率、逆行及停车等异常事件检测报警、烟雾薄雾及雨雪等低能见度报警等，非常丰富。Vantage 系统的交通流量检测精度可达 95%，平均车速检测精度高达 95%，停车报警检测精度高达 98%以上，逆行报警达 95%以上，拥堵报警达 99%以上。

7.7.1　硬件平台

由于应用环境的限制，为了满足低功耗、体积小、安装灵活等要求，这类系统一般应考虑采用嵌入式开发。硬件平台有多种选择，当前有两种主流方案。一种是以 ARM 为核的 CPU 和专用媒体处理芯片搭建，其优点是开发时间相对较短，但由于采用 ASIC，灵活性差，产品一旦定型就很难更改。另一种是采用面向媒体处理器的专用 DSP，其开发时间也不长，由于算法是软件代码，所以可以在不改变硬件的情况下不断对产品性能进行升级，重复开发成本较低。

用于视频检测的摄像机在各个方面均有一个最低要求，如分辨率最低 380 线或更高，采用固定光圈或慢调速光圈，镜头带有红外滤光片和眩光抑制，具有良好的对比度等。一般情况下使用黑白摄像机，彩色摄像机灵敏度通常是黑白摄像机的 1/4 左右，在晚上可能会导致检测性能的严重下降。在光照条件较好的情况下，可以对车辆的跟踪提供颜色信息。

摄像机安装的最佳位置取决于应用类型（数据采集、事件报警或存在检测），并且会受到环境条件的制约。一般来讲，摄像机应尽可能安装高一些，并尽可能固定在检测区域的中间位置，要尽可能避免遮挡问题。另外，尽可能将摄像机固定在稳固的立柱上，特别是长期应用。摄像机视距取决于它的安装高度和镜头，一般摄像机视距为其安装高度的 10 倍，建议安装高度不小于 10m。

7.7.2　软件设计和实现

在算法设计的过程中，需要充分考虑到交通道路上可能存在的各种噪声和干扰，并对应地采取措施进行有效的克服。例如，首先要适应各种天气情况及光照条件的挑战，包括白天和黑夜、雨天和晴天等；路面积水反光和路面抛撒物的影响；树木和护栏的阴影、车辆自身阴影的影响。对不同的天气状况，系统需要对检测精度进行适当调整。比如在恶劣天气下，为保障行车安全，不妨降低对检测精度的要求，甚至将整个检测区域都视为车辆区域。

在各种运动目标检测方法中，图像差分法由于其简单高效而被广泛采用。图像差分法也特别适用于交通流检测任务。该算法中，重构背景图像的质量对系统的检测质量影响很大。考虑到光照变化、天气变化、长周期路况改变等慢速背景变化，还需要对背景图像进行动态更新。为提高检测的可靠性，还需要对检测到的车辆进行跟踪，以防止重复计数。可以利用颜色、纹理、位置等局部信息对车辆进行匹配跟踪，跟踪算法可以利用上文介绍的 Kalman 滤波器或粒子滤波器等方法。

利用前文介绍的 Blackfin 图像和视频开发包所提供的丰富的库函数，可以快速完成软件系统的开发。其中主要用到的功能模块如下：背景图像建模、背景图像的更新、图像差分法、Kalman 滤波器等。工具包中未提供粒子滤波器对应的库函数。下面来看一下粒子滤波器的实现方法。

首先要对粒子滤波器进行初始化：

```
#define NumOfParticles 100   // 粒子的最大数目
double   gObjHist[3][16];    // 直方图
Point2D gSamplePts[NumOfParticles];
// 纯位移模型，下面分别是沿着行和列的最大位移
double   gDY = 30;
double   gDX = 30;
// 基于矩形框和掩膜初始化跟踪器
void particle_TrackInit(Image* inImage, Image* inMask, RECT inTargetBox)
{
 int     imgWidth, imgHeight, xi, yi, binIdx, i,j;
 double  maskval, scaPixel[4], pixCount, histvals[36][36][3];
 Point2D boxCenter;
 imgWidth    = inImage->width;
 imgHeight   = inImage->height;

 for (i=0;i<3; i++)                                    // 初始化目标直方图
     for (j=0;j<16; j++)     gObjHist[i][j] = 0;
 // 得到对象直方图
 pixCount = 0;
 for( xi = inTargetBox.left+1 ; xi <= inTargetBox.right ; xi++ ) {
     for( yi = inTargetBox.top+1 ; yi <= inTargetBox.bottom ; yi++) {
         maskval = getMaksVal( inMask, yi, xi);     // 掩码值
         if (maskval==0) continue;
         scaPixel = getPixelVal( inImage, yi, xi);  // 得到像素值
         for( i=0 ; i<3 ; i++ ) { // 处理每个通道

histvals[yi-(inTargetBox.top+1)][xi-(inTargetBox.left+1)][i]
   = scaPixel.val[i];
             binIdx = (int) (scaPixel.val[i]/16);
             gObjHist[i][binIdx]++;
         }
         pixCount++;
     }
 }
 for (i=0;i<3; i++) // 归一化直方图
     for (j=0;j<16; j++)         gObjHist[i][j] /= pixCount;
 // 在当前位置初始化粒子，这是一个围绕举行中心的简单高斯聚类
 boxCenter.x = (inTargetBox.left + inTargetBox.right)/2.0;
```

```
        boxCenter.y = (inTargetBox.top + inTargetBox.bottom)/2.0;
        for (i=0;i<NumOfParticles; i++){
            gSamplePts[i].x = boxCenter.x + sqrt(gDX/2.5)*gaussrand();
            gSamplePts[i].y = boxCenter.y + sqrt(gDX/2.5)*gaussrand();
        }
        srand ( time(NULL) );       // 初始化随机数生成器
        gTkFrameCount = 0;          // 重置跟踪帧计数
       }
```

然后对接下来的每帧图像进行跟踪：

```
      void particle_TrackNextFrame(Image* inImage, RECT inStartBox, TkResult
*outResult)
       {
        int boxWidth, boxHeight, boxHalfWidth, boxHalfHeight, dispHalfWin;
       int i, j, k, xi,yi, binIdx, maxX, maxY, indices[NumOfParticles];
        double cumWeightSum[NumOfParticles+1] , particleWeight[NumOfParticles];
        double sampleHist[3][16], xprime[NumOfParticles], yprime[NumOfParticles];
        double vx, vy, pixCount, bhattSsore, likelihood, sumWeight, u1, uj,
randval;
        RECT        sampleBox, targetBox, displayBox;
        double scaPixel[4];     Point2D predictPts[NumOfParticles];

        boxWidth    = inStartBox.right-inStartBox.left;
        boxHeight   = inStartBox.bottom-inStartBox.top;
        boxHalfWidth    = boxWidth/2;   boxHalfHeight   = boxHeight/2;
        // 利用匀速位置假设预测粒子位置，仅应用高斯噪声
        sumWeight = 0;
        for (i=0;i<NumOfParticles;i++){
            vx = sqrt(gDX)*gaussrand(); vy = sqrt(gDY)*gaussrand();
            // 预测粒子
            predictPts[i].x = gSamplePts[i].x + vx;
            predictPts[i].y = gSamplePts[i].y + vy;
            // 得到样本邻域窗口
            sampleBox.left  = cvRound(predictPts[i].x - boxHalfWidth);
            sampleBox.right = cvRound(predictPts[i].x + boxHalfWidth);
            sampleBox.top   = cvRound(predictPts[i].y - boxHalfHeight);
            sampleBox.bottom = cvRound(predictPts[i].y + boxHalfHeight);
            for (j=0;j<3; j++) // 初始化对象直方图
                for (k=0;k<16; k++) sampleHist[j][k] = 0;
```

```
    // 计算样本直方图
    pixCount= 0;
    for( yi = sampleBox.top+1 ; yi <= sampleBox.bottom ; yi++) {
        for( xi = sampleBox.left+1 ; xi <= sampleBox.right ; xi++ ) {
            scaPixel = getPixelVal(inImage, yi, xi);
            for (j=0;j<3; j++){     // 循环处理每个通道
                binIdx = (int)(scaPixel.val[j]/16);
                sampleHist[j][binIdx]++;
            }
            pixCount++;
        }
    }
    for (j=0;j<3; j++)              // 归一化直方图
        for (k=0;k<16; k++)     sampleHist[j][k] /= pixCount;
    // 基于似然函数计算权值
    bhattSsore=0;
    for (j=0;j<3; j++)
        for (k=0;k<16; k++)
            bhattSsore += sqrt(sampleHist[j][k]*gObjHist[j][k]);
    bhattSsore /= 3;
    likelihood = exp(8*bhattSsore);
    particleWeight[i] = likelihood;
    sumWeight += particleWeight[i];
}//end for (i=0;i<NumOfParticles;i++)
cumWeightSum[0] = 0;                // 归一化权值并计算累积和
for (i=0;i<NumOfParticles;i++){
    particleWeight[i] /= sumWeight;
    cumWeightSum[i+1] = cumWeightSum[i] + particleWeight[i];
}
// 进行加权随机重采样，重采样率与权值是成比例的
for (j=0;j<NumOfParticles;j++) indices[j] = 0;
randval = ((double)rand()) / RAND_MAX;
i=1;    u1 = randval/NumOfParticles;
for (j=1;j<=NumOfParticles;j++){
    uj = u1 + (j-1.0)/NumOfParticles;
    while (uj>cumWeightSum[i-1])   i=i+1;
    indices[j-1] = (i-1)-1;
}
for (j=0;j<NumOfParticles;j++){
```

```
        gSamplePts[j] = predictPts[indices[j]];
        xprime[j] = gSamplePts[j].x;
        yprime[j] = gSamplePts[j].y;
    }
    // 得到目标位置
    maxX = (int)fn_Round(fn_median(xprime, NumOfParticles));
    maxY = (int)fn_Round(fn_median(yprime, NumOfParticles));
    // 得到目标矩形
    targetBox.left      = (long)(maxX - boxHalfWidth);
    targetBox.right     = targetBox.left + boxWidth;
    targetBox.top       = (long)(maxY - boxHalfHeight);
    targetBox.bottom    = targetBox.top + boxHeight;
    // 得到前景目标掩模图像
    // ……
    outResult->FGMask = NULL;
    gTkFrameCount++;
}
```

第 8 章　视频编解码理论及实现

8.1　视频编码基本理论与技术

视频编解码是数字视频处理的重要内容，视频编码的目的是要减少视频序列的码率，以便能够在给定的通信信道上实时传输视频。信道带宽因应用和传输介质的不同而异，如对于使用常规电话线路的可视电话应用系统，可利用 20kbps 的视频编码；对于标准清晰度卫星广播电视信号，可利用 6Mbps 的码率。除了通信应用系统外，储存和检索也需要视频编码。不同存储介质有不同的容量和存储速率，因此需要不同的压缩量，为此需要不同的视频编解码算法。这些算法主要分为两类，第一类是基于波形的视频编码器，它能够对任意视频信号进行有效编码而无需分析视频内容；第二类是基于内容的视频编码器，它能够识别视频序列中的区域和物体并对它们进行编码。

视频图像数据有极强的相关性，也就是说有大量的冗余信息。其中冗余信息可分为空域冗余信息和时域冗余信息。通过编码技术可以最大限度地去除这些冗余信息，在保证一定质量的前提下极大地降低比特率。视频相邻帧之间具有很大的相关性，存在大量的时域冗余信息。使用帧间编码技术可去除时域冗余信息，它包括以下 3 部分。

（1）运动补偿：通过先前的局部图像来预测、补偿当前的局部图像，它是减少帧序列冗余信息的有效方法。

（2）运动表示：不同区域的图像需要使用不同的运动矢量来描述运动信息，运动矢量可以进一步通过熵编码进行压缩。

（3）运动估计：是从视频序列中抽取运动信息的一整套技术。

另外，数字视频是由图像序列组成的，因此可以充分利用静态图像压缩技术，去除大量空域冗余信息。这方面主要使用了变换编码和熵编码技术。

（1）变换编码：帧内图像和预测差分信号都有很高的空域冗余信息。变换编码将空域信号变换到另一正交空间，使其相关性下降，数据冗余度减小。

（2）量化编码：经过变换编码后，产生一批变换系数，对这些系数进行量化，使编码器的输出达到一定的位率。当然这一过程会导致图像质量下降。

（3）熵编码：是无损编码，它对变换、量化后得到的系数和运动信息，进一步压缩位码流。

本章首先介绍视频编码的基本理论和技术，然后简要介绍现有的几个视频编解码的国际标准，接下来介绍利用 DSP 实现 H.264 视频编解码的技术，最后对 ADI 提供的 H.264 编解码器及其使用进行具体介绍。

8.1.1　信源编码的信息论基础[1~3]

在视频编码或任何信号编码中，把给定信号看做随机过程的一个实现。在信源编码理论中，把随机过程视为信源。编码方法的效率取决于如何充分利用信源的统计特性。信息论的几个重要结果建立了信源编码的理论基础。这些结果建立了实现无损压缩和有损压缩所需要的最低比特率的界限。下面给出两个基本定理：无损编码定理和信源编码定理，分别用来测试无损编码系统和有损编码系统的性能。

1．无损编码定理

离散无记忆信源 X 无损编码所能达到的最小码速率为

$$\min\{R\} = H(X) + \delta \quad \text{比特/符号} \tag{8-1}$$

其中 R 是信息传输速率，$H(X)$ 是信源的熵，δ 是一个绝对量，它可以任意趋近于 0。

无损编码定理为信源传输所需的最小码速率提出了一个下界限，可以实现零编码－解码误差。在离散无记忆信源中通过对每一个符号单独编码可以对这个界限进行处理。对于有记忆信源来说，则必须对连续 N 个信源符号组成的块一次性进行编码，才能任意接近该界限。

2．信源编码定理

在有损编码方案中，能够达到的最小速率是所允许的失真度的函数。编码速率和失真度之间的关系由率失真函数（如图 8.1 所示）给出。在一给定的失真度 D 情况下，率失真函数 $R(D)$足够用来重构信源，该信源的平均失真度任意趋近于 D。对于允许的失真度 D，实际速率 R 的最小值为 $R(D)$，即

$$R \geqslant R(D) \tag{8-2}$$

函数 $R(D)$称为率失真函数，计量单位为比特/符号。注意，$R(0) = H(X)$，即率失真函数退化为无损压缩的特殊情况。

图 8.1 为一个典型的率失真函数。对于简单的信源和失真函数，率失真函数可以分析计算出。当分析方法不适用时，用计算机算法可以计算出 $R(D)$。总之，我们感兴趣的是一个压缩系统，以达到在给定的失真度下速率最低，或在给定的速率下失真度最低。

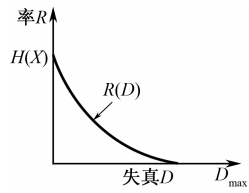


图 8.1　率失真函数图形

8.1.2　无损压缩

现在研究和运用的主要是二进制编码，它是用二进制比特序列（称为码字）表示有限字母表信源中每个可能符号的过程。所有可能符号的码字形成码书。一个符号可以对应一个或几个原始的或量化后的像素值或模型参数。因为从符号到码字的映射是一一对应的，

因此这个过程也称为无损编码。

下面介绍两种比较流行的无损编码方法。严格全面地介绍编码过程超出本书的范围，这里只是帮助读者理解无损编码的思想，以及学习如何运用它们来指导实际编码技术的设计。

1．游程编码

游程编码的思想非常简单。首先输出符号 a_i（或者数据），然后输出连续数据长度 l_i。信源编码时，先根据确定的符号 a_i 和允许误差 ε 搜索后续的符号 x_j，如果满足 $|x_j - a_i| \leqslant \varepsilon$，则游程长度加 1；如果条件不满足，则游程编码结束，并且根据后续数据重新确定新的编码符号 a_{i+1}，然后重复上述过程直至编码结束。

游程编码实际上是一种特殊的序列编码方式，利用了符号（或数据）之间的相关性进行编码，从而减少了数据描述长度，提高了编码效率。但是在一般情况下，并不单独使用游程编码对数据进行压缩，而是将其作为整体编码算法中的一种编码单元加以使用。游程编码算法在图像压缩中得到了广泛应用，连续色调静态图像压缩标准 JPEG-LS 中就采用了游程编码算法，以提高数据压缩效率。

游程编码有二进制和非二进制之分，取决于具体编码要求。对于二进制编码，就是统计连续 0 或 1 的个数。由于其特殊性，游程编码可以不输出再现符号，数据长度的交替就是符号 0 和 1 之间的交替，这样只需要约定或在开始时给出再现符号即可。

二进制游程编码在比特平面编码技术中得到了广泛的应用，在静态图像压缩标准 JPEG 和 JPEG2000 中，都使用这种游程编码作为编码单元实现二进制符号的初步压缩。为了进一步提高压缩效率，标准中还使用算数编码对游程输出进行进一步压缩。

2．算术编码

使用分组码进行编码时，如果信源的符号较少，要取得好的编码效果只有采用扩展信源进行编码。例如，二进制信源只有两个符号 0 和 1，对信源符号直接编码，每个符号要用一个比特，不可能降低信源输出符号的长度。如果使用长的符号序列对二进制信源进行扩展，可以有效提高编码效率，只要序列足够长，就可以逼近信源的熵。

从信源编码角度来看，非常希望设计一种有效编码算法对信源符号进行扩展，从而构成尽可能长的序列。尽管哈夫曼编码是一种有效的编码算法，但并不满足这种要求，因为它采用自下而上的编码方法，要求计算特定长度的所有信源序列的概率，并且构造完整的码表。理想编码的方法是：能够任意扩展序列长度，而又不重复构造码表，然后再进行编码。算数编码能够实现这样的编码要求。

与变长编码算法一样，算数编码是以信源的概率分布作为编码依据的。其基本思想是：对于给定的信源序列，计算序列的概率密度函数 $p(x^N)$ 和累加概率 $F(x^N)$，从而将信源输出序列 $x^N = (x_1, x_2, \cdots, x_N)$ 与累加概率 $F(x^N)$ 对应起来，即用有效精度的累计概率表示信源的输出序列，准确地讲，是使用概率间隔 $[F(x^N) - p(x^N), F(x^N)]$ 表示序列。然后将 $F(x^N)$ 用

$\lceil -\log p(x^N) \rceil$位精度表示，其中$\lceil \cdot \rceil$表示上取整，就是用一个有限精度的概率表示信源序列的一个码字。显然，信源发送的序列符号不同，对应的累加概率也不相同，而且当序列长度 N 一定时，在不同序列之间不会产生概率间隔的重叠，从而保证了码的唯一性。

下面简单描述算数编码的方法，在实际应用中，采用累计概率 $P(S)$表示码字 $C(S)$，符号概率 $p(S)$表示状态区间 $A(S)$，则有

$$\begin{cases} C(S,r) = C(S) + A(S)P_r \\ A(S,r) = A(S)P_r \end{cases} \tag{8-3}$$

对于二进制符号组成的序列，r=0，1。

实际编码过程是这样的：先设定两个存储器，起始时可令

$$A(\varnothing) = 1, C(\varnothing) = 0 \tag{8-4}$$

式中∅代表空集，即起始时码字为 0，状态区间为 1。每输入一个信源符号，存储器 C 和 A 就按照式（8-3）更新一次，直至信源符号输入完毕，就可将存储器 C 的内容作为该序列的码字输出。由于 $C(S)$是递增的，而增量 $A(S)P_r$ 随着序列的增长而减小，因为状态区间 $A(S)$越来越小，与信源单符号的积累概率 P_r 的乘积也越来越小。所以 C 的前面几位一般已固定，在以后计算中不会被更新，因而可以边算边输出，只需保留后面几位用作更新即可。译码也可逐位进行，与编码过程原理相同。

实际应用中，有时需要通过分类方法来进行算数编码，提高编码效率。假设时刻 n 的条件概率与前面时刻的状态有关，在此情况下可以使用分类方法简化迭代计算，而且当分类模型能够充分反映信源的统计特性时，不仅算数编码实现简单，而且编码效率也很高。

8.1.3　变换编码

通过某种正交变换可以降低信号的相关性，提高压缩效率。我们最熟悉的正交变换是傅里叶变换，但是离散傅里叶变换产生的系数都是复数形式，不利于后续的信源编码。在实际编码系统中，经常使用离散余弦变换和离散小波变换作为变换工具，将离散信源输出的数据表示为实数系数，经过量化以后得到整型系数，然后对量化后的系数进行编码。下面简单介绍离散余弦变换（DCT）。

在语音、视频编码等应用中，由于离散余弦变换算法简单、有效，因此得到了广泛的应用。离散余弦变换是离散傅里叶变换的一种特殊形式，其变换系数如下

$$\begin{aligned} a_{ok} &= \frac{1}{\sqrt{N}} \\ a_{ik} &= \sqrt{\frac{1}{N}} \cos\frac{(2k+1)i\pi}{N} \end{aligned} \tag{8-5}$$

其中 N 为数据个数，设 $x_i(i = 0,1,\cdots,N-1)$ 为原始数据，$y_i(i = 0,1,\cdots,N-1)$ 为变换得到的系数，则正、逆变换的矩阵形式为

$$\begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{N-1} \end{bmatrix} = \frac{2}{\sqrt{N}} \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \cdots & \frac{1}{\sqrt{2}} \\ \cos\frac{\pi}{2N} & \cos\frac{3\pi}{2N} & \cdots & \cos\frac{(2N-1)\pi}{2N} \\ \vdots & \vdots & & \vdots \\ \cos\frac{(N-1)\pi}{2N} & \cos\frac{3(N-1)\pi}{2N} & \cdots & \cos\frac{(N-1)(2N-1)\pi}{2N} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{N-1} \end{bmatrix} \tag{8-6}$$

$$\begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{N-1} \end{bmatrix} = \frac{2}{\sqrt{N}} \begin{bmatrix} \frac{1}{\sqrt{2}} & \cos\frac{\pi}{2N} & \cdots & \cos\frac{(N-1)\pi}{2N} \\ \frac{1}{\sqrt{2}} & \cos\frac{3\pi}{2N} & \cdots & \cos\frac{3(N-1)\pi}{2N} \\ \vdots & \vdots & & \vdots \\ \frac{1}{\sqrt{2}} & \cos\frac{(2N-1)\pi}{2N} & \cdots & \cos\frac{(N-1)(2N-1)\pi}{2N} \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{N-1} \end{bmatrix} \tag{8-7}$$

或者用矩阵表示为

$$\begin{aligned} \boldsymbol{Y} &= \boldsymbol{G}\boldsymbol{X} \\ \boldsymbol{X} &= \boldsymbol{G}^{\mathrm{T}}\boldsymbol{X} \end{aligned} \tag{8-8}$$

式中，T 表示矩阵转置，$\boldsymbol{X} = [x_0, x_1, \cdots, x_{N-1}]^{\mathrm{T}}$，$\boldsymbol{Y} = [y_0, y_1, \cdots, y_{N-1}]^{\mathrm{T}}$ 分别为原始数据和变换后的系数，而 G 为变换矩阵为

$$\boldsymbol{G} = \frac{2}{\sqrt{N}} \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \cdots & \frac{1}{\sqrt{2}} \\ \cos\frac{\pi}{2N} & \cos\frac{3\pi}{2N} & \cdots & \cos\frac{(2N-1)\pi}{2N} \\ \vdots & \vdots & & \vdots \\ \cos\frac{(N-1)\pi}{2N} & \cos\frac{3(N-1)\pi}{2N} & \cdots & \cos\frac{(N-1)(2N-1)\pi}{2N} \end{bmatrix} \tag{8-9}$$

由于图像编码是二维数据，离散余弦变换应当在水平、垂直方向分别进行，即对图像每行的数据进行一次离散余弦变换，得到一组系数，然后将每行对应的系数作为输入数据再进行垂直方向的离散余弦变换，变换过程如图 8.2 所示。变换后的系数经过量化变为整型数据，然后才能进行熵编码，实现图像数据压缩。

图 8.2　基于离散余弦变换的图像压缩编码框图

在图像压缩中，一般将图像划分为独立的图像子块分别进行离散余弦变换，子块的大小一般是 8×8 或 16×16。由于余弦函数是浮点的，离散余弦变换是不可逆变换，因此不能用于无损图像压缩。静态图像压缩国际标准 JPEG、动态图像压缩标准及语音压缩中都使用离散余弦变换去除数据之间的相关性。由于采用分块编码技术，当输出码率较小时，基于离

散余弦变换的图像压缩后的重建图像会产生明显的方块效应。

8.1.4　预测编码

预测就是从已收到的符号来提取关于未收到的符号的信息，从而预测其最可能的值作为预测值，并对它与实际值之差进行编码，达到进一步压缩码率的目的。由此可见，预测编码是利用信源的相关性来压缩码率的，对于独立信源，预测就没有可能了。

预测的理论基础主要是估计理论。估计就是用实验数据组成一个统计量作为某一物理量的估值或预测值。最常见的估计是利用某一物理量在被干扰下所测定的实验值，这些值是随机变量的样值，可根据随机量的概率分布得到一个统计量作为估值。若估值的数学期望等于原来的物理量，就称这种估计为无偏估计；若估值与原物理量之间的均方误差最小，就称为最佳估计。用来预测时，这种估计就成为最小均方误差的预测，所以也就认为这种预测是最佳的。

要实现最佳预测就需要找到计算预测值的预测函数。设有信源序列 $x_1,x_2,\cdots,x_r,x_{r+1},\cdots$。$r$ 阶预测就是由 $x_1,x_2,\cdots,x_r$ 来预测 x_{r+1}。可令预测值为

$$x'_{r+1}=f(x_1,x_2,\cdots,x_r) \tag{8-10}$$

式中 f 是待定的预测函数。要使预测值具有最小均方误差，必须确知 r+1 个变量 $(x_1,x_2,\cdots,x_r,x_{r+1})$ 的联合概率密度函数，这在一般情况下是很困难的。因而常用线性预测的方法来达到次佳的结果。线性预测就是预测函数为各已知信源符号的线性函数，即 x_{r+1} 的预测值为

$$x'_{r+1}=f(x_1,x_2,\cdots,x_r)=\sum_{s=1}^{r}a_s x_s \tag{8-11}$$

均方误差为

$$D=E(x'_{r+1}-x_{r+1})^2 \tag{8-12}$$

要求使均方误差最小时的各 a_s 值，可将式（8-11）代入式（8-12），对各 a_s 取偏导数并置零后得

$$\frac{\partial D}{\partial a_s}=-E\{(x_{r+1}-\sum_{s=1}^{r}a_s x_s)x_s\}=0 \tag{8-13}$$

只需已知信源各符号之间的相关函数即可进行运算。最简单的预测是令

$$x'_{r+1}=x_r \tag{8-14}$$

这可称为零阶预测，常用的差值预测就属这一类。高阶线性预测已在话音编码，尤其是在声码器中广泛应用。

如果信源是非平稳的或非概率性的，无法获得确切和恒定的相关函数，不能构成线性预测函数，可采用自适应预测的方法。一种常用的自适应预测方法是设预测函数是前几个符号值的线性组合，即令预测函数为

$$x' = \sum_{s=1}^{r} a_s x_{t-r-1-s} \tag{8-15}$$

再用已知信源序列来确定各系数 a_s，使对该序列所造成的均方误差 D 最小。此时的各系数 a_s 并不能保证对该信源发出的所有序列都适用，只有在平稳序列情况下，这种预测的均方误差可逼近线性预测时的最小值。随着序列的延长，各系数 a_s 可根据以后的 n 个符号值来计算，因而将随序列的延长而变更，也就是说，可以不断适应序列的变化，适用于时变的非平稳信源序列。这里简要介绍两种利用预测编码进行图像压缩的方法。

（1）矢量量化方法（VQ）。将一组采样值量化成有限个数组状态中的一个，它对声音和图像信号数据的标量量化更具有多种优点，因为它能够有效地利用量化的数据采样值间的统计相关性。因此，与标量量化相比，当重构层数固定时 VQ 提供较低的失真；或者是当失真度固定时，提供的重构层数较低。可以证明：即使数据源是无记忆的，当矢量维数增加时，VQ 能够对率失真边界进行处理。

（2）子带编码方法。其基本思想是把一个图像的傅里叶频谱分成若干个互不重叠的频带，然后对每一个子带进行反变换，得到一组带通图像。根据每一个带通图像的频率，对它们进行二次采样，并且单独用一个比特率进行编码，这个比特率要和各个子带的频率及主观视觉能力的要求相匹配。解码器通过附加上行采样和经适当过滤的子图像变形来重构原始图像。

8.2 视频编码国际标准

如果希望不同厂家生产的各种终端能够交换或接受来自公共信源（如电视广播台）的信息，就需要制定一套标准。开发一套国际标准需要来自不同国家的许多同行的合作，并需要一个能支持标准化过程和实施标准的组织。标准化组织面向电信行业创建了 ITU（国际电信联盟），面向贸易产业创建了 ISO（国际标准化组织）。ITU 交互式视频编码标准主要包括 H.261、H.263 和 H.264 等标准。ISO 下属的运动图像专家组（MPEG）定义了娱乐和数字电视的 MPEG-1、MPEG-2、MPEG-4 和 MPEG-7 等一系列标准。尽管 ISO 和 ITU 在定义广泛应用的音频和视频编码标准方面是很成功的，但在定义因特网上多媒体信号传输方面却不大成功。这方面目前由因特网工程任务组 IETF 负责，IETF 是一个大而开放的由网络设计者、经营者、厂商及设计因特网结构发展和平稳运行的研究人员组成。

视频编码标准的发展分为 3 个阶段：竞争、集中和验证。在竞争阶段，定义了标准的应用范围和需求；进一步，专家们聚在一起证明他们的最好算法。通常一旦定义了需求，标准化组织就发出一个征求建议通知，以便征求整个社会的进入。这个阶段的特点是独立地进行竞争性实验。集中阶段的目的是合作实验以便达成编码方法的一致。当对标准的第一个公共框架达成一致时，考虑如编码效率、主观质量、实现复杂度及兼容性等问题。在不同的实验室实现这个框架，并且精炼其描述，直到不同的实现达到相同的效果。在验证阶段，检查说明书是否有差错和歧义。生成一致性比特流和正确解码的生成序列。依从标

准的解码器必须把每个一致性比特流解码成正确的视频序列。

现代视频编码标准的应用主要包括以下几个方面。

（1）视频电视会议的视频编码导致了用于 ISDN 视频会议的 ITU 标准 H.261，用于在模拟电话线路传输视频会议和连接到因特网的桌面和运动终端的 H.263，以及用于宽带视频会议的 MPEG-2 视频等。

（2）用于在 CD-ROM 上存储电影及其他消费视频应用系统中的视频编码，如最初的 MPEG-1 标准广泛用于 VCD、卡拉 OK 机等地方，其码率为 1.2Mbps。

（3）用于广播和 DVD 数字视频存储视频编码的 MPEG-2 标准，码率为 2～15Mbps，以及扩展的码率范围为 15～400Mbps 的 HDTV 视频编码，应用系统包括卫星电视、有线电视、视频编辑和存储等。

（4）分离的音视频对象编码在 ISO MPEG-4 中被标准化，并应用于因特网视频、交互式视频、内容管理、专业视频、二维和三维计算机图形及移动视频通信等。

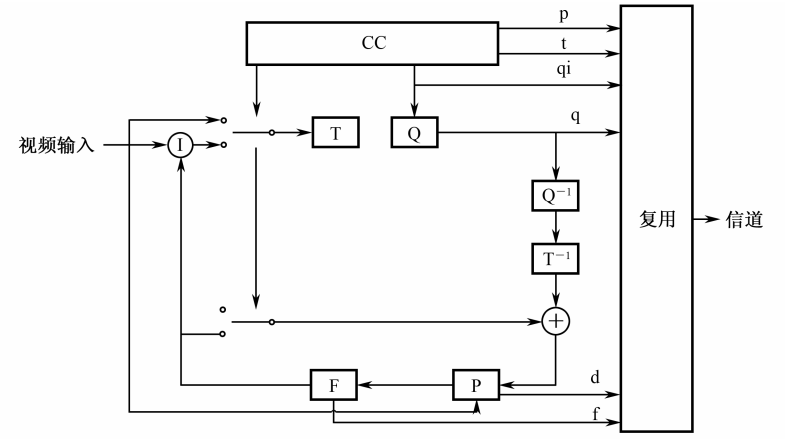
下面按照 ITU-T 视频编码标准的发展过程，主要介绍 H.261、H.263 及 H.264。

8.2.1　H.261 视频编码标准

H.261[20]是 ITU-T 为在综合业务数字网（ISDN）上开展双向声像业务（可视电话、视频会议）而制定的，速率为 64kbps 的整数倍。H.261 是最早的运动图像压缩标准，它详细制定了视频编码的各个部分，包括运动补偿的帧间预测、DCT 变换、量化、熵编码，以及与固定速率的信道相适配的速率控制等部分。H.261 仅对 CIF 和 QCIF 两种图像格式进行处理，每帧图像分成图像层、宏块组（GOB）层、宏块（MB）层、块（Block）层来处理。

如图 8.3 所示为以 4∶2∶0 采样格式处理视频的 H.261 编码器方框图。它是一种具有运动补偿的基于块的混合编码器，把图像分成 16×16 像素尺寸的宏块。一个宏块由 4 个亮度块和两个色度块（一个用于 C_r 分量，一个用于 C_b 分量）组成。H.261 对每个块使用 8×8 DCT 以减少空间冗余度，使用 DPCM 环路以利用时间冗余度，并对宏块使用单向整数像素向前运动补偿，以提高 DPCM 环路的性能。

可采用一种简单的二维环路滤波器对运动补偿运动信号进行低通滤波（图 8.3 中的方框 F）它一般会减少预测误差并降低预测图像的块效应。环路滤波器可分离为具有系数 $\left[\frac{1}{4},\frac{1}{2},\frac{1}{4}\right]$ 的一维水平和垂直函数。H.261 对 DCT 系数采用两个量化器。用步长为 8 的均匀器量化帧内模式的 DC 函数，用步长为 2～62 的接近均匀的中间踏板量化器量化帧内模式和帧间模式的 AC 系数（如图 8.4 所示）。在 $-T$～T 之间的输入称为死区，被量化为 0。除了死区之外，步长是均匀的。这个死区避免对主要会引起编码噪声的许多小的 DCT 系数进行编码。



I—变换器； Q—量化器； P—具有运动估计器和运动补偿单元的图像存储器； F—环路滤波器； CC—编码控制； p—帧内/帧间标志；	t—传送或不传送标志； qi—量化器指示； q—变换系数的量化索引； d—运动矢量； f—环路滤波器的开关切换。

图 8.3　H.261 编码器方框图

解码器主要传输每个编码宏块的两类信息：由预测误差信号变换产生的 DCT 系数（图 8.3 中的 q）和由运动估计器估计出的运动矢量（图 8.3 中的 d 和方框 P）。运动矢量的范围限制在±16 个像素内。通知解码器一个宏块和它的块是否编码及如何编码的控制数据称为宏块类型（MTYPE）和编码块模式（CBP）。如表 8.1 所示为各种宏块类型及其相应的 VLC 码字。在帧内模式中，比特流包含每个块的变换系数。作为选择项，可以标示量化器步长±2 的改变。在帧间模式中，编码器可选择只传送具有或不具有环路滤波的差分编码运动矢量。可以选择传送 CBP，以便指定需要传递变换系数的块。由于标准不规定编码器，由编码器厂商确定一种有效的编码控制(图 8.3 中的 CC)来最优地选择 MTYPE、CBP、MQUANT、环路滤波器及运动矢量（MV）。作为粗略的原则，可以选择 MTYP、CBP 及 MVD 使预测误差最小。然而，因为传送 MV 花费额外的比特，所以只当使用 MV 比不用 MV 预测误差低很多时，才传送 MV。编码图像时量化器步长是变化的，从而图像编码所需比特数不需要比编码器在两个编码帧之间所能传送的比特数更多。

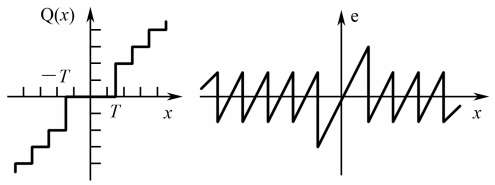


图 8.4　H.261 使用的交流 DCT 系数量化器

表 8.1　宏块类型的 VLC 表

预测	MQUANT	MVD	CBP	TOOEFF	VLC
帧内				×	0001
帧内	×			×	0000 001
帧间			×	×	1
帧间	×		×	×	0000 1
帧间+MC		×			0000 0000 1
帧间+MC		×	×	×	0000 0001
帧间+MC	×	×	×	×	0000 0000 01
帧间+MC+FIL		×			001
帧间+MC+FIL		×	×	×	01
帧间+MC+FIL	×	×	×	×	0000 01

宏块内的大部分信息是用由测试序列的统计特性导出的可变长码进行编码的。二维 DCT 系数采用游程编码方法进行编码。具体地说，量化的 DCT 系数通过“Z”型扫描转换成符号。每个符号包括在最近的一个非零系数之后被量化为零的系数个数及当前的非零系数的幅度。对于 H.261 和其他的视频编码标准，所有的 DCT 系数都转换成（游程，值）这样的符号。如果 DC 系数是非零的，那么第一个游程为零。每个符号用 VLC 进行编码。在块的最后一个非零系数之后，编码器输出一个块结束（EOB）符号。

H.261 并未规定视频解码器的能力。然而，表 8.2 中列出了 H.261 解码器必须支持的图像格式、建立视频会议通话的几个标准及交换终端之间的视频能力。在 H.320 定义的最小级别上，一个解码器必须能以 7.5Hz 的速率解码 QCIF 帧。可选的能力级别定义为以 15Hz 解码 CIF 帧。最大的级别要求以 30Hz（精确地说是 30000/1001Hz）的速率解码 CIF 帧。

表 8.2　由 H.261 和 H.263 所支持的图像格式

	Sub-QCIF	QCIF	CIF	4 CIF	16 CIF	自定义尺寸
亮度宽度（像素）	128	176	352	704	1408	<2048
亮度高度（像素）	96	144	288	576	1152	<1152
H.261		√	Opt	Still		
H.263	√	√	Opt		Opt	Opt

8.2.2　H.263 视频编码标准

H.263 是 ITU-T 为低于 64kbps 的窄带通信信道制定的视频编码标准。它是在 H.261 基础上发展起来的，其标准输入图像格式可以是 S-QCIF、QCIF、CIF、4CIF 或 16CIF 的彩色 4:2:0 亚取样图像。H.263 与 H.261 相比采用了半像素的运动补偿，并增加了 4 种有效的压缩编码模式。H.263 包括一个基线解码器，该基线解码器具有任何 H.263 解码器必须实现的

特性。另外，还定义了可选特性。以下特性是 H.263（1995）与 H.261 的主要区别。

（1）半像素运动补偿。在物体运动需要高空间分辨率的情况下，该特性可大大提高运动补偿算法的预测能力。在非整数运动矢量的情况下，用双线性内插计算预测像素。运动矢量的编码是用 3 个邻近宏块的中值运动矢量作为对该矢量的每个分量的预测（如图 8.5 所示）。如果一个宏块在图像或组块的外面，则假设其运动矢量为零。如果有两个宏块在外面，则用剩下的运动矢量作为预测。

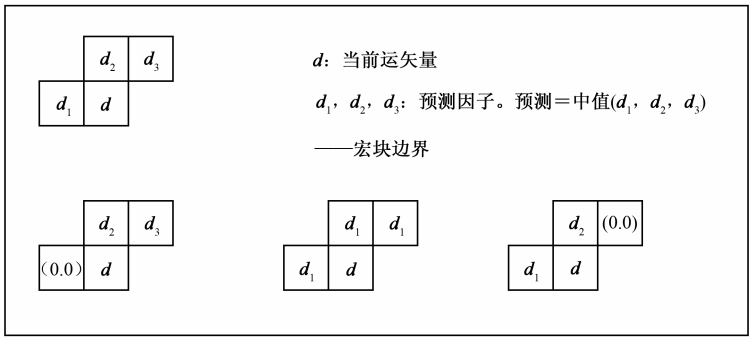


图 8.5　使用 $\boldsymbol{d}_1$、$\boldsymbol{d}_2$ 和 $\boldsymbol{d}_3$ 中值的运动矢量预测

（2）改善的可变长编码，包括三维 VLC 以提高 DCT 系数编码的效率。H.261 对符号（游程，幅值）进行编码，并在每个块的结束发送出一个 EOB 字。H.263 把 EOB 字结合到 VLC 中。要编码的事件是（最后，游程，幅值），其中“最后”指示该系数是不是块中最后的一个非零系数。

（3）在块组级及 MTYPE 和 CBP 编码中降低开销。

（4）支持更多图像格式（如表 8.2 所示）。

除了这些改善外，H.263 还提供了一组定义在标准的附件中的可选特性。

（1）无限定运动矢量。允许指向图像的外面，在摄像机运动或运动发生在图像边界的情况下可提高编码效率。对指向图像外面的运动矢量的预测信号是由重复图像的边界像素生成的。运动矢量范围扩展到[–31.5，31]。

（2）基于语法的算术编码代替可变长（哈夫曼）编码，对 P 帧平均比特率节省 4%，对 I 帧节省 10%。但是解码器计算需求增长了 50%以上。

（3）先进预测模式包括无限定运动矢量模式，提供了两个附加改善。第一个改善是重叠块运动补偿（ORMC），可用来预测图像的亮度分量，提高了预测性能并显著减少了块失真。在一个 8×8 的亮度预测块中，每个像素是由 3 个运动矢量计算得到的 3 个预测值的加权和。这 3 个运动矢量是：当前块的矢量和最靠近当前 8×8 块的两个宏块的矢量。

第二个改善是一个宏块可选择使用 4 个运动矢量，每个亮度块一个，以更好地模拟真实运动。这需要编码器决定在哪个宏块中 4 个运动矢量足以证明编码这些运动矢量所需要的额外比特是值得的。这些运动矢量也是预测来的编码（如图 8.6 所示）。

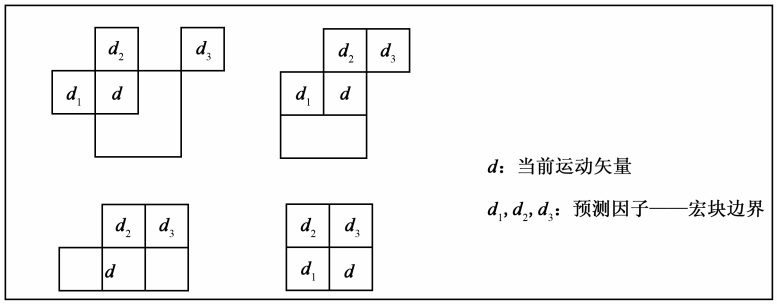


图 8.6　以先进预测模式预测运动矢量

（一个宏块中具有 4 个运动矢量：当前运动矢量 ***d*** 的分量的预测值是其预测因子的中值。）

（4）PB 图像是把双向预测图像与一般前向预测图像一起编码的一种模式。B 图像在时间上位于 PB 图像的 P 图像之前。与基于逐帧计算的双向预测对比，PB 图像使用宏块级的双向预测。在一个 PB 帧中，每个宏块的块数目是 12 而不是 6。在每个宏块中，属于 P 图像的 6 个块先传输，随后是 B 图像的块（如图 8.7 所示）。双向预测来自前一个解码的帧和当前宏块的 P 块。如在图 8.7 中所看到的，对于在 B 图像和 P 图像之间运动的情况下，这就把后向预测限制于 B 块中那些与当前 P 宏块内的像素对齐的像素（图 8.7 中的淡灰度区）。对于 B 块中的淡灰度区，通过前向和后向预测结果的平均计算预测。B 块的白色区域中的像素只用前向运动补偿进行预测。一种改进的 PB 帧模式后来被采纳，该模式去除了这种限制，能够得到常规 B 帧的效率。

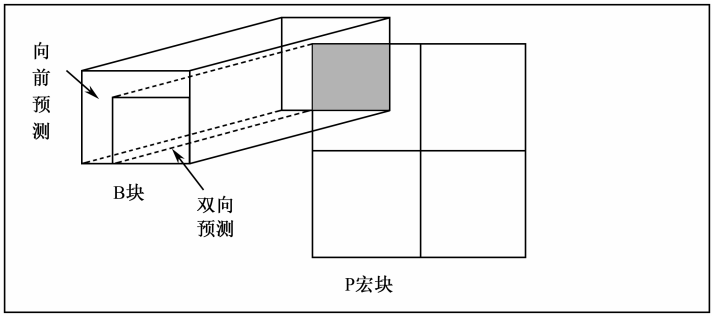


图 8.7　PB 帧中前向预测可用于所有 B 块

（后向预测只用于其后向运动矢量与当前宏块的像素对齐的那些像素。）

PB 图像对中等幅度运动的图像序列非常有效，但对快速运动、复杂运动场景或以低帧率编码时效果不是很好。由于 B 图像的图像质量不影响后续帧的编码，H.263 规定 PB 图像集的 B 图像以低于 P 图像的质量编码，对 P 块使用比响应的 B 块更小的量化器步长。PB 图像增加了编码系统的延迟，因为它们只在后面的 P 帧被摄取和处理后才允许编码器送出 B 帧的比特。这限制了它们对交互式实时应用系统的用途。

由于编码模式的数目较大，编码器的判决比 H.261 中的更复杂。H.263 最初版本制定后

进一步添加了可选模式。这里列出以前未提到的最重要的推荐模式。

（1）先进的帧内编码。帧内块用左侧的块或上方的块作为预测来进行编码，只要该块也是以帧内模式编码的。这种模式将 I 图像的编码效率提高了 10%～15%。

（2）去除块效应滤波器。在解码的 8×8 块的边界应用自适应滤波器，以减小块失真。这个滤波器也影响预测图像，它是在编码器和解码器预测环的内部实现的。

（3）补充增强信息。这种信息可用来提供给由使用 H.263 的应用系统定义的用于外部应用的标志信息。进一步，它可用于通知增强的显示能力，如帧冻结、缩放或色键。

（4）改进的 PB 帧模式。这种模式去掉了加于后向预测的限制，因此这种模式使常规的双向预测成为可能。

8.2.3 H.264 视频编码标准

H.264 是由 ISO/IEC 与 ITU-T 组成的联合视频组（JVT）制定的新一代视频压缩编码标准。1996 年制定 H.263 标准后，ITU-T 的视频编码专家组（VCEG）开始了两个方面的研究：一个是短期研究计划，在 H.263 基础上增加选项（产生了 H.263+与 H.263++）；另一个是长期研究计划，制定一种新标准以支持低码率的视频通信。长期研究计划产生了 H.26L 标准草案，在压缩效率方面与先期的 ITU-T 视频压缩标准相比具有明显的优越性。2001 年 ISO 和 MPEG 组织认识到 H.26L 潜在的优势，随后开始组建联合视频组（JVT），主要任务就是将 H.26L 草案发展为国际标准。在 ISO/IEC 中该标准命名为 AVC （先进视频编码），作为 MPEG-4 标准的第 10 个部分；在 ITU-T 中正式命名为 H.264 标准。

H.264 视频编解码器的基本结构与早期的编码标准（H.263、MPEG4 等）相似，都是由运动补偿、变换、量化、熵编码、环路去块效应滤波器等功能单元组成的。H.264 标准的改进主要体现在各功能模块内部。H.264 的重大改进表现在以下几个方面。

（1）高精度的基于 1/4 像素精度的运动预测。

（2）多种宏块划分模式。每个宏块（16×16 像素）的亮度分量有 7 种分区方法：16×16、16×8、8×16、8×8、8×4、4×8、4×4。

（3）多帧预测。在帧间编码时，可选 5 个不同的参考帧。

（4）整数变换。采用基于 4×4 像素块的整数变换代替 DCT 变换。

（5）帧内预测编码。采用多种设计合理的帧内预测模式，大大降低了 I 帧的编码率。

（6）H_264/AVC 支持两种熵编码方法，即 CAVLC（基于上下文的自适应可变长编码）和 CABAC（基于上下文的自适应算术编码）。CAVLC 的抗差错能力比较高，而编码效率比 CABAC 低；CABAC 编码效率高，但需要的计算量和存储容量更大。

（7）网络适配层 NAL（Network Abstraction Layer）为视频编码层提供一个与网络无关的统一接口，使视频编码数据能适应不同的网络应用环境。

下面介绍这些改进的相关细节。

1．4×4 整数变换

以前的标准，如 H.263 或 MPEG-4，都是采用 8×8 的 DCT 变换。H.26L 中建议的整数变换实际上接近于 4×4 的 DCT 变换，整数的引入降低了算法的复杂度，也避免了反变换的失配问题，4×4 的块可以减小块效应。而 H.264 的 4×4 整数变换进一步降低了算法的复杂度，相比 H.26L 中建议的整数变换，对于 9b 输入残差数据，由以前的 32b 降为现在的 16b 运算，而且整个变换无乘法，只需加法和一些移位运算。新变换对编码的性能几乎没有影响，而且实际编码略好一些。

2．基于空域的帧内预测技术

视频编码是通过去除图像的空间与时间相关性来达到压缩的目的。空间相关性通过有效的变换来去除，如 DCT 变换、H.264 整数变换等；时间相关性则通过帧间预测来去除。这里所说的变换去除空间相关性，仅仅局限在所变换的块内，如 8×8 或 4×4，并没有块与块之间的处理。H.263+与 MPEG-4 引入了帧内预测技术，在变换域中根据相临块对当前块的某些系数做预测。H.264 则是在空域中，利用当前块的相邻像素直接对每个系数做预测，更有效地去除相邻块之间的相关性，极大地提高了帧内编码的效率。

H.264 基本部分的帧内预测包括 9 种 4×4 亮度块的预测、4 种 16×16 亮度块的预测和 4 种色度块的预测。

3．运动估计

H.264 的运动估计具有 3 个新的特点：1/4 像素精度的运动估计；采用 7 种大小不同的块进行匹配；前向与后向选取多参考帧。

H.264 在帧间编码中，一个宏块（16×16）可以被分为 16×8、8×16、8×8 的块，而 8×8 的块被称为子宏块，又可以分为 8×4、4×8、4×4 的块。总体而言，共有 7 种大小不同的块做运动估计，以找出最匹配的类型。与以往标准的 P 帧、B 帧不同，H.264 采用了前向与后向多个参考帧的预测。半像素精度的运动估计比整像素运动估计有效地提高了压缩比，而 1/4 像素精度的运动估计可带来更好的压缩效果。编码器中运用多种大小不同的块进行运动估计，可节省 15%以上的比特率（相对于 16×16 的块）。

运用 1/4 像素精度的运动估计，可以节省 20%的码率（相对于整像素预测）。多参考帧预测方面，假设为 5 个参考帧预测，相对于一个参考帧，可降低 5%～10%的码率。以上百分比都是统计数据，不同的视频因其细节特征与运动情况不同而有所差异。

4．熵编码

H.264 标准采用的熵编码有两种：一种是基于内容的自适应变长编码（CAVLC）与统一的变长编码（UVLC）结合；另一种是基于内容的自适应二进制算术编码（CABAC）。CAVLC 与 CABAC 根据相邻块的情况进行当前块的编码，以达到更好的编码效率。CABAC 比 CAVLC 压缩效率高，但要更复杂一些。

5．去块效应滤波器

H.264 标准引入了去块效应滤波器，对块的边界进行滤波，滤波强度与块的编码模式、运动矢量及块的系数有关。去块效应滤波器在提高压缩效率的同时，改善了图像的主观效果。

H.264 分为 7 种不同的类（Profile）：基线类、主类、扩展类、高级类、高级 I/O 类、高级 4:2:2 类和高级 4:4:4 类，分别代表不同的技术限制和算法集合。其中基线类的使用是不收版权费的。

与以上技术升级相对应，H.264 有如下主要优点。

（1）在相同的重建图像质量下，H.264 比 H.263+和 MPEG-4（SP）减小 50%码率。

（2）对信道时延的适应性较强，既可工作于低时延模式以满足实时业务，如会议电视等；又可工作于无时延限制的场合，如视频存储等。

（3）提高网络适应性，采用“网络友好”的结构和语法，加强对误码和丢包的处理，提高解码器的差错恢复能力。

（4）在编/解码器中采用复杂度可分级设计，在图像质量和编码处理之间可分级，以适应不同复杂度的应用。

当然，新技术在带来了较高压缩比的同时，也大大提高了算法的复杂度，因而对处理器性能提出了更高的要求，特别是对嵌入式开发提出了更大的挑战。

8.2.4 其他视频编码标准

除上述 ITU-T 的视频压缩标准外，还有一些标准也比较流行，如 MPEG-4、AVS、WM9 等。H.264 也称为 MPEG-4 AVC，而目前业内所说的 MPEG-4 一般是指 SP（简级）或 ASP（先进的简级），主要针对低码率应用，如因特网上的流媒体、无线网的视频传输及视频存储等，其核心类似于 H.263。MPEG-4 SP 和 H.263 有很多相似的地方。然而，这两个标准之间也有显著的不同，主要表现在：码流结构和头信息、熵编码的部分码表、编码技术的一些细节。MPEG-4 ASP 较 SP 增加了一些技术，主要有 1/4 像素精度的运动估计、B 帧、全局运动矢量（GMV），因而压缩效率得以提高。

AVS 是由我国自主制定的音/视频编码技术标准，主要面向高清晰度电视、高密度光存储媒体等应用。AVS 标准以当前国际上最先进的 MPEG-4 AVC/H.264 框架为基础，强调自主知识产权，同时充分考虑了实现的复杂度。相对于 H.264，AVS 的主要特点如下。

（1）8×8 的整数变换与 64 级量化。

（2）亮度和色度帧内预测都是以 8×8 块为单位，亮度块采用 5 种预测模式，色度块采用 4 种预测模式。

（3）采用 16×16、16×8、8×16 和 8×8 这 4 种块模式进行运动补偿。

（4）在 1/4 像素运动估计方面，采用不同的四抽头滤波器进行半像素插值和 1/4 像素插值。

（5）P 帧可以利用最多 2 帧的前向参考帧，而 B 帧采用前后各一个参考帧。

Window Meida 9(WM9)是微软公司开发的新一代数字媒体技术。一些测试表明，WM9 的视频压缩效率比 MPEG-2、MPEG-4 SP 及 H.263 高很多，而与 H.264 的压缩效率相当。

目前，H.261 与 H.263 在视频通信中广泛应用，成熟的产品已经很多。H.263 与 H.261 相比，增加了若干选项，提供了更灵活的编码方式，压缩效率大大提高，更适应网络传输。H.264 标准的推出，是视频编码标准的一次重要进步，它与现有的 MPEG-2、MPEG-4 SP 及 H.263 相比，具有明显的优越性，特别是在编码效率上的提高，使之能应用于许多新的领域。

8.3　基于 Blackfin 的 H.264 视频编解码系统设计

H.264 视频编码性能的大幅提高是以复杂度的成倍增加为代价的，这也使得 H.264 在实时视频编码及传输应用中面临着巨大的挑战。要满足图像压缩的实时性要求，就需要对现有的 H.264 编解码器进行优化，对嵌入式系统更是如此。下面讨论基于 ADI Blackfin 533/561 EZ-Kite Lite 评估板的 H.264 开发。在软件实现中，需要针对 DSP 处理器的特点对 H.264 标准算法进行代码级优化，以达到实时性要求[56,57]。

为了在 DSP 平台上实现实时编解码，需要优化程序的设计。优化流程如下：

（1）在 PC 上进行算法验证和评估、优化程序的流程设计和数据结构设计。

（2）将程序代码移植到 Blackfin 处理器。在 VisualDSP++集成开发环境里进行编译，删除 PC 平台相关的代码，添加 DSP 平台相关的代码。

（3）进行基于 DSP 平台的优化操作。设置速度优化的编译参数，进行 C 语言级的优化，然后用汇编指令改写最耗时的函数，通过使用专用的向量指令和并行指令减少函数的执行时间。

1．在 PC 上实现并优化解码器程序

根据系统要求，选择 ITU 的某个版本（如 Jm8.5）的基线类作为标准算法软件。ITU 的参考软件 JM[56]是基于 PC 设计的，故可取得较高的编码效果。将视频编解码软件移植到 DSP 时，需要考虑 DSP 的系统资源，其中主要的考虑因素是系统空间的限制——包括程序空间和数据空间。所以，需要对原始的 C 代码进行评估，这就需要对所移植的代码有较全面地了解。如图 8.8 所示是 H.264 编码器算法系统框图。

接下来需要确定编码算法中运算量较大、耗时较长的部分。Visual Studio 中自带的 Profile 分析工具显示：帧内与帧间编码部分占用了整体运行时间的 60%以上。其中运动估计更是占用了其中较多的时间。所以移植与优化的重点应在运动估计部分。为此对代码结构进行如下调整。

（1）大幅删减不必要的文件和函数。对于基线类、单一参考帧的情况，很多文件和函数都可以删减，以减小最终生成代码的尺寸。其中包括有关 B 帧、SI 片、SP 片和数据分割、分层编码、权值预测模式、CABAC 编码模式等不需要的特性的冗余程序代码。此外，还可以删除 top_pic、bottom_pic 等与场有关的全局变量与局部变量、分层编码、多切片分割及

FMO、与场编码/帧场自适应编码/宏块自适应编码有关的预测、参考帧排序、输入/输出及解码器缓存操作等。还可以删除随机帧内宏块刷新模式和权值预测模式等相关的冗余代码（如使编码器采用 NAL 码流而非 RTP 格式）。同时删除 rtp.c、sei.c 中包含的一些辅助编码信息（并不编入码流中）。如果不用，也可以删除 leaky_bucket.c 用于计算泄漏缓存器的参数。具体模块是否要保留还应根据具体应用需求来确定。

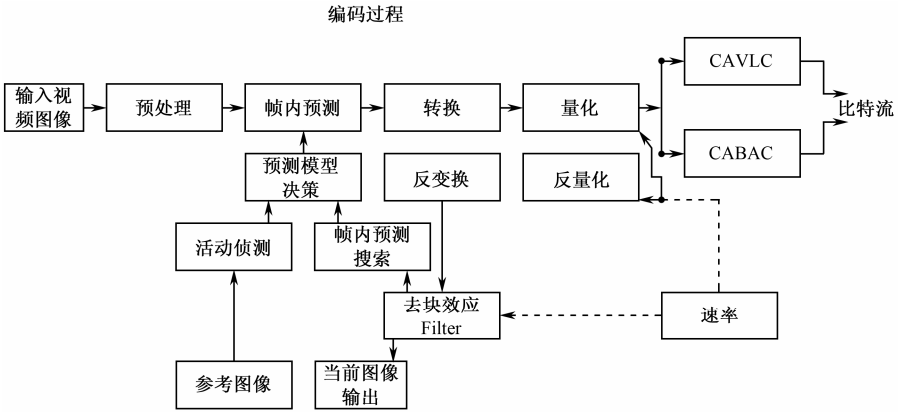


图 8.8　H.264 编码器算法系统框图

（2）配置函数的改写。由于 JM 的系统参数配置是通过读取 encoder.cfg 文件来实现的，故可将参数配置由读取文件改为通过初始化集中赋值函数来实现。这样既减少了代码量，又减少了对有限内存空间的占用和读取时间，提高了编码器整体的编码速度。例如，定义为 int 型的变量 input->img_height 就可直接改写为 input->img_height = 288（CIF 格式）。

（3）去除冗余的打印信息。

为了调试与算法改进的方便，JM 保留了大量的打印信息。为了提高编码速度，减少存储空间消耗，这些信息完全可以删掉，如大量的跟踪信息和编码数据统计文件。如果 lor.dat 和 stat.dat 仅需在 PC 上调试时使用，也没必要移植到 DSP 平台上，相关代码完全可以去除。但是，调试时所需的基本信息（如码率、信噪比、编码序列等）则应保留参考。

（4）算法细节的改进。如 JM8.5 中每次处理一个切片时都要分配内存、读取其中信息、再释放内存，这里就应该合理安排内存空间的分配和释放以提高速度；还有可将 I 帧、P 帧分别独立解码，宏块解码也按预测模式和预测方向分成不同的解码模块，以省去中间的重复判断，提高解码速度；以及优化 CAVLC 码表的查询方法等。

通过调整可使得代码的结构、容量更加精简，从而为接下来在 DSP 上的移植做好准备。

2．程序移植

将优化的 PC 程序移植到 Visual DSP 下，使其能够初步运行，主要要解决的是语法规则和内存分配等问题。

（1）去除编译环境不支持的函数。主要是去除某些与时间相关的函数、将文件操作修改为读取文件数据缓存的操作、删除 SNR 信息收集等 DSP 平台实现所不需要的代码。还要

注意：函数的声明、数据结构的类型要符合 DSP 的 C 语言格式。

（2）添加与硬件相关的代码。该代码包括系统初始化、输出模块代码、中断服务程序和码速率控制程序等。

（3）配置 LDF 文件。代码和数据都非常大，SRAM 放不下，因而会导致链接错误。刚开始最好把所有程序和数据都放在 SDRAM 里，链接就不会有问题了。堆栈和堆情况类似，都先放到 SDRAM 中。这样在一开始就可以得到一个可以正确运行的程序，然后再逐步优化以提高其速度。

（4）Malloc 问题的解决。DSP 下 Malloc 是一个需要解决的问题。如果动态申请内存，就算可以运行，其结果往往也是不对的。所以，最好进行静态分配，可用数组的形式分配。

移植完毕后，即可实现基于 Blackfin 处理器的 H_264 编码。但此时速度尚达不到实时解码的要求，还需要进行进一步的优化。

3．基于 DSP 平台的优化

基于 DSP 平台在 VisualDSP++开发环境下对代码进行优化分为系统级优化、C 语言级优化和汇编级优化。

（1）系统级优化。打开编译器中的优化开关，设置为速度最优化；打开自动内联开关；打开过程间优化开关；并使用 VisualDSP++编译器的 PGO 优化编译技术。

编写链接描述文件，将经常用的数据存储在片内存储器上，如 CAVLC 熵解码的码表；启用指令 Cache 和数据 Cache，设置好启用 Cache 机制的指令地址和数据地址。

减少对片外存储器的访问次数。对于经常访问的片外存储器区域，设置 Cache 为使能状态，并可设置 Cache 锁定，防止缓存的数据被替换，减少 Cache 未命中的机率。

（2）C 语言级优化。移植与优化的重点在运动估计部分，算法实现中采用了菱形（DS）搜索法。DS 算法可采用两种搜索模板，分别是 9 检索点大模板 LD-SP 和 5 检索点小模板 SDSP。菱形搜索过程如图 8.9 所示。搜索时先用大模板，当最小块误差 SAD 点出现在中心点处时，再换用小模板。这时 5 个点中具有最小 SAD 者若为中心点，则该点即为最优匹配点，然后结束搜索，否则继续以此点为中心搜索。

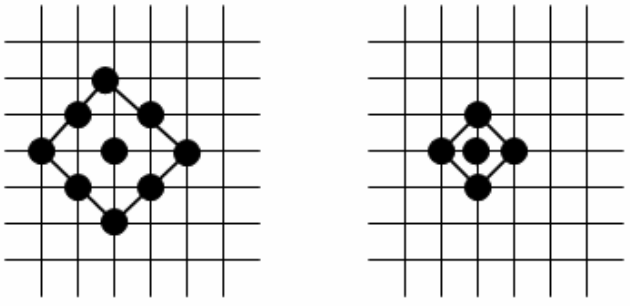


图 8.9　菱形搜索示意图

针对 DSP 的特点和相关的硬件指令，设计时可对代码进行如下优化。

① 对程序结构进行调整，改写不适合 DSP 执行的语句，提高代码的并行性。

② 使用宏：将较短、执行单一、调用次数多的函数改为宏。

③ 循环优化：将 C 语言中的 for 循环打开，排流水线，提高并行性。

④ 将计算表格化：将运行时计算的参数做成查找表，将运行计算转化为编译运算。

如在量化和反量化程序中进行移位位数的处理时，可先计算出所有可能的值，而后来的运算就可以通过查表直接得到数值。

⑤ 浮点数定点化：因为 Blackfin 不支持浮点运算，但原程序却是浮点运算格式，所以必须改成定点运算，修改后的执行速度也会加快很多。

⑥ 尽量用逻辑运算代替乘除运算：由于乘除指令执行时间远远大于逻辑移位指令，尤其是除法，应尽量用逻辑移位运算来代替乘除运算。

⑦ 尽量少进行函数调用：对一些小函数，最好用适当的内联函数将其直接写入主函数中进行替代；对于一些调用不多的函数，也可直接写入主函数内，这样可减少不必要的操作以提高速度。

⑧ 对能用较短数据类型表达的数据改用较短数据类型表达，如原定义为 int 类型的 4×4 逆整数变换的输入数据，实际上可以定义为 short 类型。

⑨ 减少判断转换。

⑩ 尽量静态分配内存。

⑪ 调用系统提供的丰富的内联函数。

此外，为充分发挥 DSP 的运算能力，还必须从其硬件结构出发，最大限度利用它的 8 个功能单元，使用软件流水线尽量让程序无冲突地并行执行。也可将最耗时的函数抽取出来，用线性汇编改写，从而最大限度的利用 DSP 的并行性。

（3）汇编级优化。将最耗时的函数用汇编语言改写，可以充分利用 Blackfin 处理器 SIMD 结构的优点和硬件上的并行性，在一个指令周期内执行多个操作，减少函数执行需要的指令周期。最耗时的函数主要有宏块解码函数、逆整数变换函数、去块效应滤波函数、滤波门限计算函数等。

汇编级优化通常遵循以下原则。

① 使用寄存器资源。比如 Blackfin561 提供了 8 个 32 位数据寄存器及一系列地址寄存器。使用寄存器代替局部变量时，若局部变量用来保存中间结果，那么用寄存器代替局部变量可省掉很多访问内存的时间。

② 使用硬件循环代替软件循环。Blackfin 处理器有专用的硬件支持两级嵌套的零开销硬件循环。用硬件循环代替软件循环可避免堵塞流水线，提高速度。

③ 使用并行指令和向量指令。使用并行指令和向量指令，可以充分利用 Blackfin 处理器的 SIMD 系统结构的优点和内部硬件资源的并行处理优点，减少指令执行次数和提高指令执行效率。使用 1 条并行指令同时执行 2 条或 3 条非并行指令。向量指令可以同时对多个数据流进行相同的加工操作。

④ 使用视频处理指令等专用指令。视频处理应用可以使用 Blackfin 处理器专用的视频

处理指令，提高执行效率。其他专用指令包括求最大值、最小值、绝对值、CUP 等。通过使用这些指令能大大提高代码的执行速度。如用 int 型（32 位）访问两个 short（16 位）型数据时，可将其分别放在 32 位寄存器的高 16 位和低 16 位字段。这样数据读取效率可以提高 1 倍，从而减少内存访问次数。

⑤ 合理存放程序和数据段。把反复调用的程序段（如 DCT 变换）放在片内程序存储区中，把频繁用到的数据段（如编码表）放在片内数据存储器中，把不常用的程序和数据段放在片外存储器。避免对程序或数据进行不必要的反复搬移。

⑥ 合理使用内外存储器。比如 BF561 片内只有 256KB 的存储空间，因此当前帧、参考帧和当前帧的重建帧都必须放到片外存储器，压缩码流若被主机读取，也可放至片外。其他数据如程序代码、全局变量、VLC 码表、各编码模块产生的中间数据等均可放至片内。

⑦ DMA 的使用。由于 CPU 访问片外存储器的速度通常要比访问片内慢几十倍，片外数据的传输通常成为程序运行时的瓶颈。这样，即使代码效率很高，流水线也会因为等待数据而被严重阻塞。解决这一问题的有效方法是用 DMA 传送数据。程序逐个宏块进行编码，在编码当前宏块的同时，先由 DMA 将下一个宏块的数据、用到的参考帧数据由片外传送至片内；当前宏块做完运动补偿后，DMA 又将重建后的宏块由片内传送至片外。这样 CPU 只对片内数据进行操作，从而使流水线可以顺利进行，而压缩码流按逐个码字有时间间隔地写入，可由 CPU 直接写至片外。

汇编级优化的一个典型例子是 SAD 优化，利用 Blackfin 提供的视频操作指令 SAA，只需要使用一条指令就可完成 4 字节的绝对差运算，在同一时钟周期可并行处理运算和更新地址内容，大大节约了时间。实验表明，采用该指令可以使 SAD 的处理速度提高近 50 倍。测试结果如表 8.3 所示。SATD 的计算也可采用汇编改写，用到了芯片的专用视频操作指令和并行、向量操作指令。

表 8.3　优化前、后函数所占时钟周期数对比表

优 化 函 数	优化前（执行指令数）	优化后（执行指令数）
SAD（16×16 宏块）	4791	88
SATD（16×16 宏块）	4863	318

另外，1/4 像素插值滤波函数对像素矩阵进行 1/4 像素插值操作。先用六阶滤波器进行 1/2 像素插值，然后用线性内插法进行 1/4 像素插值。1/2 像素 b 计算方法为

$$b = \text{round}\left(\left(E - 5F + 20G + 20H - 5I + J\right)/32\right) \qquad (8\text{-}16)$$

如图 8.10 所示，E、F、G、H、I、J 是整数像素，b 是 G 和 H 之间的 1/2 像素。

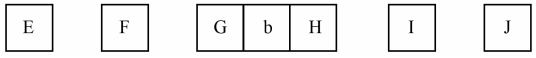


图 8.10　1/2 像素内插

像素亮度值为 unsigned char 类型，先利用并行指令可以在一个指令周期内将 8 像素的亮度值读到寄存器，然后利用视频专用指令将 4 字节解包到一个寄存器对（R1:0 或 R3:2）

中去，利用向量指令在一个周期内进行两次乘加操作。通过视频专用指令、向量指令和并行指令的使用，减少了指令运行所需的指令周期数。

相关实验证明，经过用 ADSP-BF561 汇编语言改写的对应函数的优化程序经调试运行后，DCT、IDCT 部分效率提高约 15 倍，去块滤波部分效率提高 6～7 倍，对模块中的其他部分也同样取得了良好的优化结果，说明其优化工作可以达到良好的效果。ADI 公司宣布其可以在 600 MHz 的 BF533 处理器上实现 D1（720×576）格式的视频实时解码器。

此外，ADI 公司还提供了 H.264 基线类编码器和解码器库函数，极大地方便了开发者的使用，加速了产品面市的速度。

8.4 ADI 提供的 H.264 视频编码实现

8.4.1 H.264 基线编码器概述

H.264 基线编码器（也称为 MPEG4 第 10 部分编码器）库是一个软件组件，它将视频内容编码为 H.264 基线类视频码流。它使用灵活，能够接收视频捕获设备（如摄像机）的实时视频捕获结果，或者是 PC、嵌入式系统中的视频文件。编码器每次处理一帧图像并为该帧生成基础码流。它包含可 C 语言调用的、可中断的一组 API，使得库函数能够与应用系统进行无缝集成。输入视频格式是 YUV420 或 UYVY422 ITU-R 656 或 UYVY422 逐行原始的格式。该编码器在帧边界级别上是可重入的。帧中间的可重入需要由 RTOS 或系统开发者提供。

H.264 基线类编码器主要用于以下应用：视频电话、视频会议、视频监控、手机、数字硬盘录像机（DVR）、手持媒体播放器、个人视频录像机（PVR）、编码变换等。该编码器支持以下特性：

（1）采用基线类，级别为 3.0。

（2）支持多种分辨率。

① 在 BF561 下支持实时的 D1 分辨率。NTSC 制是 720×480 每秒 30 帧；PAL 制是 720×576 每秒 25 帧。

② 在 BF533 下支持半 D1 分辨率，或每秒 15 帧的 D1 分辨率。

③ 在降低帧率时可以达到高于 D1 的分辨率。

（3）多通道编码，对不同分辨率和缩放可以分别配置。

（4）支持的输入格式包括 YUV420 或 UYVY422 或 UYVY422 原始输入。

（5）预处理特征（支持 UYVY422 和 YUV420 输入）。

① 将 UYVY422 转换为 YUV420。

② 可选的去交错滤波器，以支持交错输入。

③ 水平和垂直方向的简单 2:1 下采样。

④ 水平和垂直方向的简单 4:1 下采样。

（6）用户还可以以回调函数形式使用自己的预处理软件模块。

（7）支持感兴趣区域（ROI）。

（8）支持 I 帧和 P 帧。

（9）可伸缩的搜索引擎，以平衡编码质量和 MIPS。

（10）快速的帧内/帧间预测模式确定。

（11）1/4 像素运动差值。

（12）环路去块效应滤波器。

（13）可配置的场景变换检测。

（14）可配置的 CBR 码率控制。

（15）通过固定 QP 的 VBR 码率控制。

（16）支持不同的 Cache 配置。

H264 编码过程概述：

宏块是编码的基本处理单元。每个宏块是一个 16×16 亮度采样块和两个 8×8 的色度采样块。图像按照光栅扫描的顺序处理，从上到下、从左到右扫描 16×16 个元素，如图 8.11 所示。每个宏块可以编码为“内部”或“非内部”。宏块或者使用时间预测残差，或者使用空间预测残差。时间预测残差宏块称为宏块间或非内部的，空间预测残差宏块称为宏块内（内部）的。对于预测的情况，编码器选择最好的预测模式（具有最小残差能量的）。编码器以固定的频率（每个 GOP 尺寸）编码整帧图像，或者检测到输入视频源中有场景变化时。内部编码宏块编码的整个帧称为 I 帧或关键帧，具有前向宏块间编码的帧称为 P 帧。

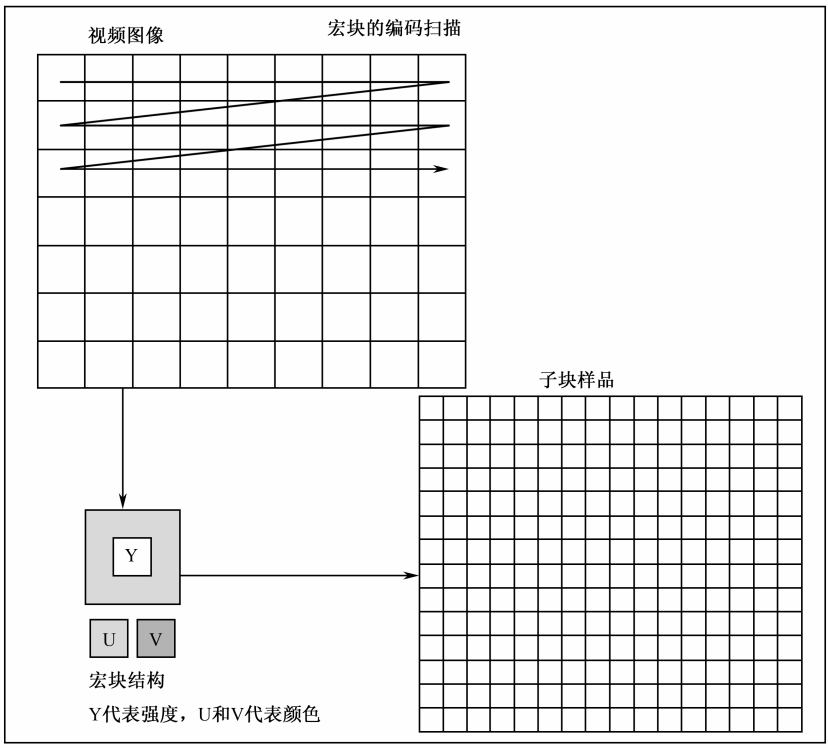


图 8.11　宏块的编码扫描

编码器核心处理部分经过优化，最大程度地从 Blackfin 处理器的内部 L1 内存执行。这是实现编解码器的关键因素。视频图像数据通过内存 DMA 在 L3 内存和 L1 内存间移动，并且与编码器的执行构成流水线结构以得到最优性能。

8.4.2 H.264 基线编码器库的使用

本节介绍如何简单快速地将 H.264 基线编码器模块[44]集成到应用程序代码中。关于每个 API 的详细介绍，参考后面的章节。

1．H.264 基线编码器集成到应用程序代码中的步骤

这里假设支持多个编码器实例。

（1）声明 video_codec_objects，其类型为 vidcodec_t。每个编码器实例需要声明一个视频编解码器对象。

（2）为编码器库函数分配 L3 内存，用于位流缓冲区、L3 状态内存和临时内存。这个过程为每个编码器都要处理一次。

（3）为内存组 A 和 B 分配 L1 内存，用于编码器库函数。

（4）初始化 video_codec_object 的 API 成员，如宽度、高度、帧速率等。

（5）调用函数 function h264_encode_init 来初始化编码器对象。应用程序需要将 video_codec_object 结构的指针传递给它，分配编码器需要的内存块，包括 ADIMemMap、位流缓冲区和输出 NAL 列表指针。

（6）调用函数 h264_encode_init_me_subpel1 为亚像素运动搜索指定算法 1。

（7）编码器现在已经准备好进行编码了。输入视频帧准备好后，调用函数 h264_encode 进行编码。编码成功后，编码器返回 inframe 指针给调用者，否则返回 0。

（8）使用输出的 pnals->nallist 来将编码器压缩后的数据取出。

（9）当编码器不再需要时，调用函数 h264_encode_close 释放所有系统资源。

要调用 H.264 基类编码器库函数，系统需要提供以下缓冲区。

（1）video_codec_object。该缓冲区最好分配在 L1 片上内存中，如果没有足够的 L1 内存，该块可以分配在 L2/L3 高速交换内存中。video_codec_object 的类型是 vidcodec_t 结构。每个 video_codec_object 保存一组会话特定的变量，用于编码一个视频通道。缓冲区尺寸至少要 0x524（1316）字节。

（2）L3 状态内存。这些缓冲区分配在 L3 内存中，可以是任意的 SDRAM 内存组。该内存块保存状态变量（编码时的使用是跨帧的），因此是特定于应用程序创建的每个编码器。编码器生成的参考帧要保存在该缓冲区。编码器用到的其他 L3 状态内存也在该缓冲区。缓冲区大小应该是：CIF 约为 0.37MB；D1 约为 1.40MB；1.3M 像素约为 4.73MB；5M 像素约为 16MB。该内存的尺寸在 adi_h264e_codec.h 中定义为常量 ADI_H264E_CODEC_L3_B3_STATE_SIZE。

（3）L1A1 临时内存。这些缓冲区最好分配到 L1 内存的内存组 L1A。如果 L1A 内存不足，可以分配到 L2/L3 高速缓冲内存。该内存用做临时内存，只在帧编码过程中使用。该内存可以在不同编码器实例间使用。缓冲区尺寸依赖于编码宽度。缓冲区尺寸是：CIF 为 2212 字节；D1 为 4512 字节；1.3M 像素为 4576 字节；5.0M 像素为 16212 字节。该内存的尺寸定义为常量 ADI_H264E_CODEC_L1A1_TEMP_SIZE。

（4）L1A0 临时内存。这些缓冲区的分配和使用原则同 L1A1 临时内存。缓冲区尺寸是：CIF 为 552 字节；D1 为 1104 字节；1.3M 像素为 2112 字节；5.0M 像素为 3912 字节。该内存的大小定义为常量 ADI_H264E_CODEC_L1B0_TEMP_SIZE。

（5）L1B1 临时缓冲区。这些缓冲区的分配和使用原则同 L1A1 临时内存，但是使用内存组是 L1B。内存大小为：CIF 为 2816 字节，D1 为 5760 字节，1.3M 像素为 11136 字节，5M 像素为 20736 字节。内存大小定义为 ADI_H264E_CODEC_L1B_TEMP_SIZE。

（6）L1B0 临时内存。这些缓冲区尽量分配到内存组 L1B。它们是可选的，只有当使用预处理特性时才被分配。预处理特性用于以下编码器设置：编码器输入格式是 UYVY422（color_format = CCIR422），这将导致颜色转换预处理被激活；输入格式是 UYVY422 或 YUV420 并且缩放和/或去交错被使用。该内存要分配在 L1B 中。应用程序要保证 L1B 有足够的内存。该内存作为临时内存使用，只在帧编码过程中使用，可以在不同编码器实例间共用。缓冲区尺寸是 4096 字节，与分辨率无关。该内存尺寸定义为常量 ADI_H264E_CODEC_L1B0_TEMP_SIZE。

（7）输出位流缓冲区。该缓冲区分配在 L3 内存中。编码器将编码后的位流发送到该缓冲区。缓冲区尺寸依赖于系统的设计，但是至少要能够保存一帧图像的编码结果。经验数据是一帧的最大尺寸大约是：比特率/帧速率×12。

（8）输入帧缓冲区。缓冲区分配在 L3 内存中。编码器从中读取输入视频采样值。为了获取实时编码的最佳性能，建议将该缓冲区分到两个 SDRAM 内存组中以实现乒乓操作，这样当编码器从一个内存组的视频帧缓冲区读取数据时，视频捕获设备驱动器可以同时填充位于另一个内存组的输入帧缓冲区。这确保了 EBIU 的最优化性能。每个内存组中输入视频帧缓冲区的数目是系统设计的一个方面，但是建议在每个 SDRAM 内存组中至少分配一帧。一个输入视频帧的大小依赖于输入视频格式，由 ccir_width×ccir_height 给定。

（9）堆栈。堆栈分配在 L1 中间变量内存中。典型的栈使用量大约是 364 字节。

下面介绍内存分段的情况。

在 LDF 文件中需要声明以下内存分段，包括指令内存和数据内存两部分。

指令内存中代码内存分段及其尺寸如表 8.4 所示。对于代码内存分段的放置有两种策略：一是只用 SRAM，没有 Cache；一是 ICache，有 16K 的 I-Cache。BF533 和 BF561 下对应的代码内存分段放置如表 8.5 所示。

表 8.4　BF533 和 BF561 的代码内存分段和尺寸

段　　名	尺寸（字节）	位　　置		
		首选	次选	再次选
adi_fast_prio0_code	8098	L1 SRAM	—	—
adi_fast_prio1_code	5640	L1 SRAM	L2 Cache	L3 Cache
adi_fast_prio2_code	13862	L1 SRAM	L2 Cache	L3 Cache
adi_fast_prio3_code	14450	L1 SRAM	L2 Cache	L3 Cache
adi_fast_prio4_code	7524	L1 SRAM	L2 Cache	L3 Cache
adi_slow_prio0_code	2920	L2 Cache	L3 Cache	L3
adi_slow_prio1_code	1732	L2 Cache	L3 Cache	L3
adi_slow_prio2_code	2696	L2 Cache	L3 Cache	L3

表 8.5　BF533 和 BF561 的代码内存分段放置方案

段　　名	BF533		BF561	
	<1>	<2>	<1>	<2>
adi_fast_prio0_code	L1 SRAM	L1 SRAM		L1 SRAM
adi_fast_prio1_code	L1 SRAM	L1 SRAM		L1 SRAM
adi_fast_prio2_code	L1 SRAM	L1 SRAM		L2 Cache
adi_fast_prio3_code	L1 SRAM	L1 SRAM		L2 Cache
adi_fast_prio4_code	L1 SRAM	L1 SRAM		L2 Cache
adi_slow_prio0_code	L3	L3 Cache		L3 Cache
adi_slow_prio1_code	L3	L3 Cache		L3 Cache

数据内存分段包括 L1 数据内存组 A、L1 数据内存组 B 和 L3 数据内存 3 部分。表 8.6 中分段应该放置到 L1 数据内存组 A 或 L2/L3 高速缓存内存分段中，表 8.7 中分段要放置到 L1 数据内存组 B 或 L2/L3 高速缓存内存分段中。分段名 prioX 中 X 表示优先级，值越小优先级越高。最后，分段 adi_slow_prio0_temp 应该放入 L3 数据内存。

表 8.6　L1 内存组 A 数据分段

段　　名	尺寸（字节）	位　　置		
		首选	次选	再次选
adi_fastb0_prio0_temp	4436	L1 SRAM	—	—
adi_fastb0_prio0_r	3364	L1 SRAM	L2 Cache	L3 Cache
adi_fastb0_prio0_rw	540	L1 SRAM	L2 Cache	L3 Cache
adi_fastb0_prio1_temp	6088	L1 SRAM	L2 Cache	L3 Cache

表 8.7　L1 内存组 B 数据分段

段　　名	尺寸（字节）	位　　置		
		首选	次选	再次选
adi_fastb1_prio0_temp	4112	L1 SRAM	—	—
adi_fastb1_prio0_r	1992	L1 SRAM	L2 Cache	L3 Cache
adi_fastb1_prio0_rw	884	L1 SRAM	L2 Cache	L3 Cache
adi_fastb1_prio1_temp	2904	L1 SRAM	L2 Cache	L3 Cache
adi_fastb1_prio2_rw	172	L1 SRAM	L2 Cache	L3 Cache

表 8.8 给出了一种在不同策略下 BF533 和 BF561 的数据内存分段放置方案。

表 8.8　BF533 和 BF561 的数据内存分段放置方案

段　　名	BF533		BF561	
	<1>	<2>	<1>	<2>
adi_fastb0_prio0_temp	L1 A SRAM	L1A SRAM	L1A SRAM	L1ASRAM
adi_fastb0_prio0_r	L1 A SRAM	L1A SRAM	L1A SRAM	L1ASRAM
adi_fastb0_prio0_rw	L1 A SRAM	L1A SRAM	L1A SRAM	L1ASRAM
adi_fastb0_prio1_temp	L1 A SRAM	L3 Cache	L1A SRAM	L3 Cache
adi_fastb1_prio0_temp	L1B SRAM	L1B SRAM	L1B SRAM	L1BSRAM
adi_fastb1_prio0_r	L1 B SRAM	L1B SRAM	L1B SRAM	L1BSRAM
adi_fastb1_prio0_rw	L1 B SRAM	L1B SRAM	L1B SRAM	L1BSRAM
adi_fastb1_prio1_temp	L1B SRAM	L3 Cache	L1B SRAM	L3 Cache
adi_fastb1_prio2_rw	L1 B SRAM	L3 Cache	L1B SRAM	L3 Cache
adi_slow_prio0_temp	L3	L3	L3	L3

需要注意的是，以下分段必须要放置到 L1：adi_fast_prio0_code、adi_fastb0_prio0_temp、adi_fastb1_prio0_temp。编码器函数的使用者必须确保所需这些分段在 L1 SRAM 内存所需的尺寸在 LDF 文件中被分配。所有 prio0 数据分段中的缓冲区需要执行 DMA 操作。将这些分段放到 L1 SRAM 可以避免编码器的异常和效率下降。

2．内存 DMA 的使用和性能

编码器库对输入（L3 到 L1）和输出（L1 到 L3）都使用 MDMA 通道。用户必须使用以下经验法则以获取编码器库的最优性能。它确保了对 EBIU 带宽的最佳使用，以及编码器库使用中 MDMA 的更好的性能。

（1）DMA 流量控制寄存器应该设置为 0x6F0。

（2）MDMA 对 SDRAM 内存访问的优先级应该高于内核。EBIU_AMGCTL 中的 CDPRIO 位必须要设置。

（3）为编码器库使用 MDMA 控制器 0 和通道 0。

（4）PPI 控制寄存器应该配置为使用 PACK_EN 和 DMA32（32 位 DMA 只存在于有 32 位 DMA 控制器的处理器中）。

（5）对于支持 32 位 DMA 控制器的处理器，PPI DMA 应该使用字（WORD）DMA 配置模式。

（6）前面介绍的输入帧缓冲区、输出码流缓冲区和 L3 状态内存最好分配到不同的 SDRAM 内存组，以提高 EBIU 访问性能。

编码器使用的 MDMA 带宽（对于 D1 输入）大约是 121MB/s。DMA 带宽大约是 3000B/s。

H.264 基线编码器数据移动带宽计算如表 8.9 所示。

表 8.9　H.264 基线编码器数据移动带宽计算

D1	121Mbytes			
QVGA	2.7Mbytes			
	计算 D1		计算 QVGA	
CCIR 输入	20.736	L3 到 L1	4.608	L3 到 L1
运动估计	84.564	L3 到 L1	18.792	L3 到 L1
参考缓冲	15.552	L1 到 L3	3.456	L1 到 L3
运动估计	0.648	L1 到 L3	0.144	L1 到 L3

3．SDRAM 中的 PPI 输入帧缓冲区

尽管 H.264 基线编码器库每次处理一个帧，并且不管理 PPI 输入帧缓冲区，它期望 PPI 输入帧缓冲区有一个特殊的结构。为了帮助编码器库获取最少的 MIPS 消耗，PPI 输入帧缓冲区应该分到两个 SDRAM 内存组，并且当编码器读取一个帧时，PPI DMA 应该不允许向同一个内存组中的帧写数据。如图 8.12 所示为 SDRAM 中的 PPI 输入帧缓冲区管理建议。

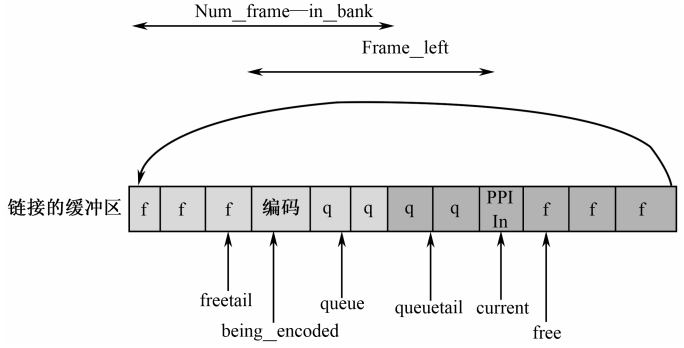


图 8.12　SDRAM 中的 PPI 输入帧缓冲区管理建议

浅色区域代表编码内存组，深色区域代表 PPI 输入内存组。Encode 区域代表正在编码的帧缓冲区，PPI In 代表正在进行 PPI 写入的缓冲区。q 代表等待编码的缓冲区，f 代表 PPI 可写入的缓冲区。

Being_encoded 指针可以移动到深色区域，只要 frame_left 大于 1；指针 current 不能移动到浅色区域；如果尚未完成编码内存组，但是 PPI 输入内存组用完了，新来的帧将覆盖 PPI 输入内存组的最后一帧，这叫做掉帧；当编码器完成编码内存组，而且 PPI 驱动程序完成了 PPI 输入内存组，就交换这两个内存组。

为了进一步了解编码器库的使用，下面介绍其中的主要函数和数据结构。

8.4.3　H.264 基线编码器 API 介绍

编码器库主要 API 函数有：h264_encode_init、h264_encode、h264_encode_close 和 h264_encode_init_me_subpel1/subpel2。下面分别介绍这些函数及其用到的重要数据结构。

1．h264_encode_init

```
int h264_encode_init ( vidcodec_t *pvc, ADIMemMap *pmemblk,
                       ubyte *bitstream, naltable_t *pnals);
```

该函数为一个编码会话初始化编码器实例对象，设置所有内部参数和指针。每个编码器实例对象与一个编码会话关联。参数 pvc 是编码器实例指针，pmemblk 是内存映射对象指针，bitstream 是编码后位流缓冲区指针，pnals 是 Nal 表指针。Nal 表描述了 NAL 单元的指针和长度。NAL 单元是 H.264 位流中的基本语法结构。NAL 表定义如下：

```
typedef struct { unsigned char *ptr; int numbytes; } nallist_t;
typedef struct { nallist_t nallist[12]; int numnal; int grosslength;
} naltable_t;
```

内存块数据结构 ADIMem 定义如下。应用程序需要确保内存块是字对齐的。

```
typedef struct {
    unsigned int nSize;              // 该结构大小
    ADIVersion nVersion;             // 该结构的版本
    void *pMem;                      // 该内存块的位置，必须是字（word）对齐
    unsigned int nMemSize;           // 该内存块的字节数
    unsigned int nFlag;              // 保存该内存块信息的寄存器
} ADIMem;
```

nFlag 字段定义如下：位 0～1 指定是 L1、L2 还是 L3；位 2～4 指定内存分组，如 L3_BANK2 或 L1A1_BANK 等；位 5～6 指定内存块类型为 TEMP_MEM、STATE_MEM 或 CONST_MEM；位 7 对应内存 Cache 配置。

内存映射定义如下，它在实例创建时传递给编码器以验证内存，并允许编码器配置其内部存储空间的管理。

```
typedef struct {
    unsigned int nSize;              // 本结构的大小
    ADIVersion nVersion;             // 本结构的版本
```

```
    unsigned int nNumMemBlocks;      // 编码器需要的内存块数目
    ADIMem *pMemBlocks;              // 编码块指针
} ADIMemMap;
```

编码器状态结构 vidcodec_t 中以下成员需要为每个编码器实例初始化。其中某些参数可以在帧间进行改变。

以下参数可以每帧发生变化。

（1）framerate：帧率，NTSC 是 2～30，PAL 是 2~25，–1 代表默认值为 25。

（2）bitrate：比特率，800000 即 800kbps。0 代表 VBR，–1 代表默认值为 1000kbps。

（3）frame_slice_num：每帧分片数，取值范围为 1～8，可以逐帧修改该值。

（4）keyframe_dist：关键帧距离，即 I 帧周期，典型值为 15～90。

（5）disable_deblocking_filter_idc：在分片边界上是否使能去块效应滤波器。

（6）slice_alpha_c0_offset_div2：去块效应滤波器α 调整强度。

（7）slice_beta_offset_div2：去块效应滤波器β 调整强度。

（8）frame_type：帧类型，–1 代表强制一帧为 I 帧，–2 代表重置速率控制。

（9）search_complexity：搜索复杂度，包括搜索范围、运动估计特性和阈值。

（10）sc_detect_flag：场景变化检测标志。

（11）rc_control：速率控制器参数。

（12）pos_x 和 pos_y：编码器实例开始进行编码的水平和垂直偏移量。

（13）fixedQP：VBR 中固定量化参数（QP）。0~51 是真实的 QP，–2 代表 QP 从 0 开始每帧递增，每 52 帧后再重置为 0。

以下是只能在初始化阶段设置的参数：

（1）video_standard：视频标准，0 是 NTSC，1 是 PAL，2 是 NTSC ACTIVE，3 是 PAL ACTIVE，4 是 RAW422 PROG。

（2）width：编码帧宽度，最小 64，编码器会将其截为 16 的倍数。

（3）height：编码帧高度，最小 64，编码器会将其截为 16 的倍数。

（4）vid_capture_width：视频捕获宽度，最小 64，必须是 4 的倍数。

（5）vid_capture_height：视频捕获高度，最小 64，必须是 4 的倍数。

（6）vid_capture_frate：捕获视频的帧频，有效范围是 1～30。

（7）color_format：颜色格式，1 代表 CCIR422，0 代表 I420。

（8）scaling：输入视频的伸缩模式，0 代表无伸缩，1 代表视频缩一半，2 代表垂直缩一半，3 代表水平垂直缩一半，4 代表水平垂直缩为 1/4。

（9）splitframe：视频帧分割标志，0 代表不分割，在一个内核编码一帧图像；1 或 2，将一帧的编码分到两个内核，coreA 对应 1，coreB 对应 2。

（10）splitheight：视频帧分割的高度。

（11）field_pic_flag：逐行还是隔行（现在不支持）。

（12）profile_idc：66 代表基线类，77 代表主类（现在不支持）。

（13）ccir_deinterlace：2 和 1 代表用算法 2 和 1 实现去交错 CCIR422 输入，0 代表禁止

去交错 CCIR422 输入。该参数只用于逐行编码模式（interlace_flag=0）。

（14）chk_for_dma_err：遇到 DMA 错误时，0 代表编码器将挂起，1 代表编码器将返回错误消息。

（15）bframecnt：B 帧数目，如该值为 0，表示不支持 B 帧。

（16）mbquant_flag：0 代表帧级别的量化参数调整，1 代表宏块级别的量化参数调整（现在不支持）。

（17）cabac_flag：0 代表禁止 CABAC 编码，1 代表使能 CABAC 编码（现在不支持）。

（18）enable_constant_pic_id：0 代表 pic_parameter_set_id 在流中可改变，1 代表 pic_parameter_set_id 在流中总是为 0。当使用 RTP/UDP 传输时，可以做到与 VLC 播放器的互操作。

（19）enable_chksum：代表是否生成检验数。

（20）enable_aspect_ratio：代表是否将画面比例放到流中。

（21）pixel_aspect_x 和 pixel_aspect_y 指定画面纵横比。

（22）preproc_api_used：是否使用编码器提供的预处理特性。

（23）enable_idr：指定 IDR NAL 单元只对第一个 I 片段有效，还是对所有 I 片段有效。对每个 IDR NAL，idr_pic_id 从 0 开始递增，最大值是 255，然后重置为 0。可通过读取视频编码器实例结构的 nal_unit_type 获取 IDR NAL 单元。

2．h264_encode

```
ifrm_t* h264_encode (vidcodec_t *pvc, ifrm_t *inframe,
                     ubyte *bitstream, naltable_t *pnals);
```

该函数对 inframe 指定的视频帧编码，并将得到的位流写入 L3 中 bitstream 指定的缓冲区。参数 pvc 指向编码器实例；inframe 指向携带输入视频帧信息的结构；bitstream 指向编码后位流缓冲区，缓冲区必须字对齐；pnals 指向 Nal 表，它描述了 NAL 单元的指针和长度，NAL 单元是 H264 位流的基本语法结构。

成功的话函数返回 inframe 指针给调用者，否则返回 0。

3．h264_encode_close

```
ifrm_t* h264_encode_close (vidcodec_t *pvc);
```

该函数释放系统资源。参数 pvc 指向编码器实例对象。

4．h264_encode_init_me_subpel1

```
void h264_encode_init_me_subpel1 (vidcodec_t *pvc);
```

函数配置编码器使用算法一进行亚像素运动估计。该算法是亚像素域改良的运动搜索实现，但最消耗 MIPS。参数 pvc 指向编码器实例对象。

5．h264_encode_init_me_ subpel2

```
void h264_encode_init_me_subpel2 (vidcodec_t *pvc);
```

函数配置编码器使用算法二进行亚像素运动估计。该算法复杂度较低，消耗较少的MIPS，但是对具有很多细节和运动的视频输入产生 1%～4%的额外比特。参数 pvc 指向编码器实例对象。

6．h264_pre_process_config

```
int h264_pre_process_config (vidcodec_t *pvc, void *ptPreProc);
```

函数配置编码器使用应用程序定义的预处理模块。该函数要对每帧图像在调用 h264_encode 前调用。参数 pvc 指向编码器实例对象，ptPreProc 指向预处理配置结构 tConfigPreProc。

```
typedef struct {
    unsigned int nSize;       // 本结构的大小
    ADIVersion nVersion; // 本结构的版本
    tDMA* (*fpVideoPreProc) (unsigned char *pYUV420Buf, void *pArg);
    void *pArg;
    tDMA *pMemDMA;
} tConfigPreProc;
```

fpVideoPreProc 是预处理模块的回调函数，pArg 是预处理模块的内部结构，pMemDma 指向 DMA 配置结构。DMA 配置结构定义如下：

```
typedef struct{
    unsigned int nSize;
    ADIVersion nVersion;
    int nMemDMA;
    tDMAConfig* pDMAConfig;
} tDMA;
```

nMemDMA 是 MDMA 块的数目，最大值为 MAX_MEM_DMA_BLOCKS；pDMAConfig 是为每个内存块建立 DMA 的参数，定义如下：

```
typedef struct{
    unsigned int nSize;
    ADIVersion nVersion;
    unsigned int isDMAConfigModified;    // 上次回调后 DMA 配置是否改变了
    unsigned char *pSrc;                 // 指向源内存块
    unsigned char *pDst;                 // 指向目标内存块
    short DMAConfigSrc;                  // 源 MDMA 配置字
    short DMAConfigDst;                  // 目标 MDMA 配置字
```

```
    short nXcntSrc;                    // 源 X 方向的字节数
    short nYcntSrc;                    // 源 Y 方向的字节数
    short nXStrideSrc;                 // 源 X 方向步长
    short nYStrideSrc;                 // 源 Y 方向步长
    short nXcntDst;                    // 目标 X 方向的字节数
    short nYcntDst;                    // 目标 Y 方向的字节数
    short nXStrideDst;                 // 目标 X 方向步长
    short nYStrideDst;                 // 目标 Y 方向步长
} tDMAConfig;
```

以上介绍的是 ADI 提供的针对 Blackfin 的 H.264 基线编码器及其使用方法，对应的解码器[45]具有类似的内容。详细内容请参考 H.264 基线解码器的使用文档。

第 9 章　视频时空滤波及实现

高清数字电视是整个电视产业发展的重心之一，液晶电视和等离子电视等的销量已经远远超过传统 CRT 电视。面对相对高额的费用，消费者对产品必然会提出更高更挑剔的要求，其中画质将是决定高清数字电视产品这个高端市场成败的重要因素。相对于传统电视，家电厂商对高清数字电视画质的重视程度更高，但是却发现数字电视画质的提升比传统显像管电视代价要大得多。一方面，显示面积的增大导致很多被忽略的问题显现出来；另一方面，显示机制的改变又带来很多新的问题，使数字电视画质提升变成一项系统工程。其中改动最容易，效果也最明显的就是数字电视视频处理系统（如图 9.1 所示）。这主要得益于超大规模集成电路制造成本的降低，以及数字视频信号处理技术的成熟。

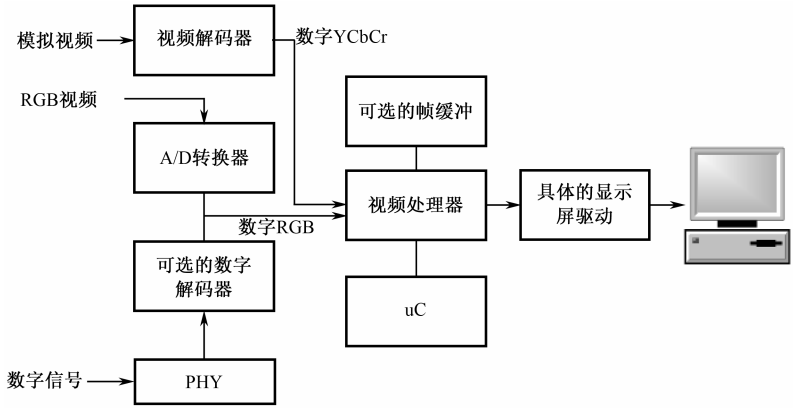


图 9.1　数字电视视频处理系统

数字电视是以像素为单位逐行显示视频图像的，因而其视频信号处理的核心工作就是将隔行信号转换为逐行信号，并进行视频缩放。这两个功能通常由一些专用芯片来完成，称为视频处理器。随着消费者对数字电视画质的要求越来越高，视频处理器除了实现基本功外，还需要加入更多画质增强技术。这些技术包括去交错、错编辑帧补偿、亮色分离补偿、瞬时亮度/色彩增强、主动色彩管理、自适应颜色/对比度增强、自适应三维噪声抑制和像素加速（Pixel Boost）技术等。

其中视频信号去交错技术由于其处理效果十分明显而被普遍采用。数字电视需要以逐行的方式显示隔行信号（如 PAL、NTSC），很多人可能会觉得只需要把隔行信号的两场图像合并起来就可以了。这样做是可行的，但是由于隔行信号的两场内容可能存在时间差，对于运动较为剧烈的视频信号，这种简单的合并图像将产生严重的锯齿，如图 9.2 所示。大

部分软解压和低端处理器使用的是倍线方法，即直接将隔行信号图像中的每一条线显示成同样的两条线。这种方法能适应大部分的视频，但是对于图像中很细的横线将产生闪烁。因此，高端视频处理器采用的是基于运动矢量的自适应去交错算法，利用前后若干场图像内容运动量的大小计算得到当前要显示的图像。

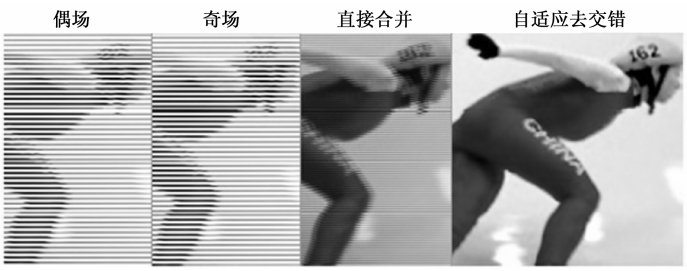


图 9.2　直接合并两场信号使运动图像产生锯齿

视频信号去交错技术是一种典型的视频时空滤波技术。本章介绍视频时空滤波的基本理论、技术及实现问题。

9.1　视频时空滤波技术

本节介绍运动补偿滤波的理论[13]，该理论是图像噪声滤波、复原、标准转换和高清晰度重构方法的基础，运动补偿也是帧间压缩方法的一个主要部分。本节将具体的运动模型结合到视频采样和重构的理论中。通过本节介绍，读者将理解时空滤波为何优于帧内滤波。

9.1.1　运动轨迹模型

我们假设时间上的强度变化是由符合简化模型的运动造成的。一般的运动其轨迹是任意的，处理起来有相当大的难度。因此，实际应用中主要采用一些简化模型处理整体恒速和加速运动等特殊情况。对这些情况我们可以推导出沿着运动轨迹作用的时空滤波器频率相应。

1．任意运动轨迹

首先定义沿任意运动轨迹的运动补偿滤波。要注意的是，在绝大多数情况下，对于 t 时刻存在的每个图像 (x_1,x_2)，滤波器的时间区域内的帧上及 τ 范围而言可以定义一个不同的运动轨迹 $c(\tau;x_1,x_2,t)$。把滤波器在 (x_1,x_2,t) 上的输出定义为

$$y(x_1,x_2,t)=Fs_1(\tau;c(\tau;x_1,x_2,t)) \tag{9-1}$$

式中，$s_1(\tau;c(\tau;x_1,x_2,t))=s_c(c(\tau;x_1,x_2,t),\tau)$ 是表示沿着运动轨迹通过 (x_1,x_2,t) 的输入图像的一维信号，F 是一个沿运动轨迹工作的一维滤波器。F 可以是线性或非线性的。

只有当每帧各个像素运动轨迹互相平行时，其运动补偿器才是平移不变的。当整体恒

速（瞬时）运动时，情况就是这样的。在整体加速（瞬时）运动或在任何空间变化运动的情况下，运动补偿滤波均将导致一种平移变化滤波器。显而易见，运动补偿滤波的时空频率域分析只有在线性、平移不变的条件下进行。

2．恒速整体运动

可以将沿着一条恒速运动轨迹、进行一次线性、平移不变的滤波操作表示为

$$\begin{aligned} & y(x_1,x_2,t) \\ &= \iiint h_1(\tau)\delta(z_1-v_1\tau,z_2-v_2\tau)s_c(x_1-z_1,x_2-z_2,t-\tau)\mathrm{d}z_1\mathrm{d}z_2\mathrm{d}\tau \\ &= \int h_1(\tau)s_c(x_1-v_1\tau,x_2-v_2\tau,t-\tau)\mathrm{d}\tau \\ &= \int h_1(\tau)s_1(t-\tau;x_1,x_2,t)\mathrm{d}\tau \end{aligned} \tag{9-2}$$

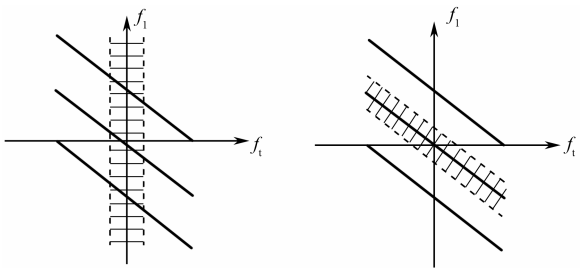
式中，$h_1(\tau)$ 为沿运动轨迹所用的一维滤波器的脉冲响应，v_1 和 v_2 为运动矢量的分量。该时空滤波器的脉冲响应可以表示为

$$h(x_1,x_2,t)=h_1(t)\delta(x_1-v_1t,x_2-v_2t) \tag{9-3}$$

对式（9-3）进行三维傅里叶变换，可以得到运动补偿滤波器频率响应为

$$\begin{aligned} & H(F_1,F_2,F_t) \\ &= \iiint h_1(t)\delta(x_1-v_1t,x_2-v_2t)\mathrm{e}^{-\mathrm{j}2\pi(F_1x_1+F_2x_2+F_tt)}\,\mathrm{d}F_1\mathrm{d}F_2\mathrm{d}t \\ &= \int h_1(t)\,\mathrm{e}^{-\mathrm{j}2\pi(F_1v_1+F_2v_2+F_t)t}\mathrm{d}t \\ &= H_1(F_1v_1+F_2v_2+F_t) \end{aligned} \tag{9-4}$$

图 9.3 中的阴影部分表示当 v_1=0 时，以及当 v_1 与输入视频信号相匹配的情况下，投影到 (F_1,F_2) 平面上的运动补偿滤波器的频率响应区域。v_1=0 时的情况与无运动补偿的纯时间滤波一致。对该图的观察可以说明，当 v_1 与输入视频信号的速度相一致时，可以得到适当的运动补偿。



（a）无运动补偿，v_1=0　　（b）有运动补偿

图 9.3　无运动补偿与有运动补偿滤波器频率响应

3．加速运动

与整体恒速运动情况类似，我们希望运动补偿滤波器区域与信号区域相匹配。可以证明，在加速运动情况下，STS（窗口信号帧传递区域）区域随 t_0 变化，我们期望这种情况下

的运动补偿滤波器具有一个随时间变化的频率响应，即它是一个线性时间变化滤波器。

如果用 $h_1(t)$ 表示沿一条加速运动的轨迹在 t_0 时刻的一次一维滤波器操作，那么运动补偿时空滤波器的随时间变化脉冲响应 $h_{t_0}(x_1,t)$ 可以表示为

$$\begin{aligned} h_{t_0}(x_1,t) &= h_1(t)\delta(x_1-(v_1+a_1t_0)t-\frac{a_1}{2}t^2) \\ &= h_1(t)\delta(x_1-v_1't-\frac{a_1}{2}t^2) \end{aligned} \tag{9-5}$$

其随时间变化的频率响应可以写为

$$\begin{aligned} H_{t_0}(f_1,f_t) &= \iint h_1(t)\delta(x_1-v_1't-\frac{a_1}{2}t^2)\mathrm{e}^{-\mathrm{j}2\pi f_1x_1}\mathrm{e}^{-\mathrm{j}2\pi f_tt}\mathrm{d}x_1\mathrm{d}t \\ &= \int h_1(t)\mathrm{e}^{-\mathrm{j}2\pi f_1(v_1t+\frac{a_1}{2}t^2)}\mathrm{e}^{-\mathrm{j}2\pi f_tt}\mathrm{d}t \end{aligned} \tag{9-6}$$

适当地选择参数 v_1 和 a_1，这种随时间变化的频率响应区域将与带有整体加速度运动的输入视频信号区域相匹配。

9.1.2　运动补偿滤波

运动补偿方法一般假设任意运动轨迹 $c(\tau;x_1,x_2,t)$ 上的像素灰度级变化主要是由于噪声引起的。对每个像素的运动轨迹用低通滤波，即可有效减少图像静态区域和运动区域内的噪声。运动补偿滤波器有以下差别：运动估算方法；滤波器区域，如时间及时间—空间；滤波器结构。

图 9.4 表示在时空采样图像序列中运动轨迹的概念及其估算值。假设需要对图像序列（共 $N=2M+1$ 个帧：$k-M,\cdots,k-1,k,k+1,\cdots,k+M$ ）中第 k 帧进行滤波，则第一步是估算每个像素 (n_1,n_2) 上的运动轨迹 $c(l;n_1,n_2,k)$。函数 $c(l;n_1,n_2,k)$ 返回对应于终点 (x_1,x_2) 的坐标，它对应于第 k 帧的像素 (n_1,n_2)。图 9.4 中实线描绘了当 N=5 帧的条件下的离散运动轨迹。在对轨迹进行估算时，位移矢量通常根据虚线表示的帧 k 来估算。一般来讲，该轨迹通过子像素位置，其上的强度可以通过空间或时空插入来确定。运动补偿时空滤波器区域 $S_{n_1,n_2,k}$ 可定义为沿运动轨迹以像素为中心的空间邻域的并集。时间滤波中，滤波区域 $S_{n_1,n_2,k}$ 恰与运动轨迹 $c(l;n_1,n_2,k)$ 一致。显然运动补偿时空滤波效率与运动估算准确度密切相关。

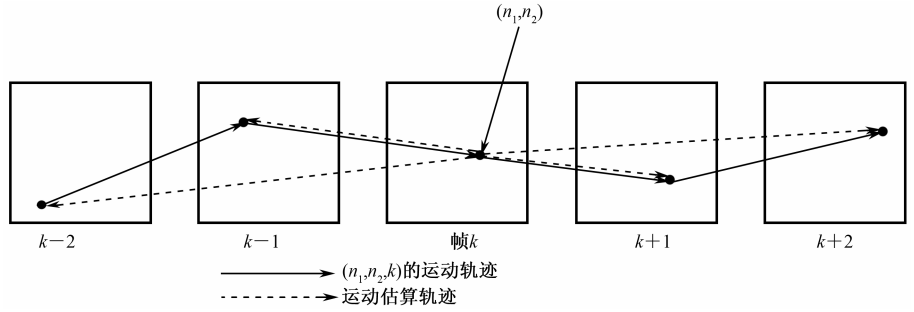


图 9.4　运动轨迹（N=5）的估算

各种滤波技术，从平均算法到自适应滤波，都可用于运动补偿滤波器。如果运动估算准确，沿运动轨迹对图像强度直接平均可有效减少噪声。但实际上运动估算很难准确。因此估算的运动轨迹上的图像强度可能对应于不同的图像内容，直接时间平均可能产生虚像。时空滤波情况下还可能存在区域内空间变化，此时自适应滤波器结构可能更为重要。此时可以考虑采用自适应加权平均滤波器，它能够抑制产生不一致分离物的影响，同时通过集中保持相似的图像强度从而得到有效的滤波。

1．时空自适应 LMMSE 滤波

运动补偿自适应 LMMSE 滤波器是由 Lee 及 Kuan 等人提出的沿保护空间滤波器向时空域的扩展。其中局部空间统计被其时空相应量所代替。在 (n_1,n_2,k) 上像素值估算如下

$$\hat{s}(n_1,n_2,k)=\frac{\sigma_s^2(n_1,n_2,k)}{\sigma_s^2(n_1,n_2,k)+\sigma_v^2(n_1,n_2,k)}[g(n_1,n_2,k)-\mu_g(n_1,n_2,k)]+\mu_s(n_1,n_2,k) \tag{9-7}$$

式中，$\mu(n_1,n_2,k)$ 和 $\sigma_s^2\ (n_1,n_2,k))$ 分别表示整体均值及响应信号的方差。滤波器将分别涉及 LMMSE-ST 或 LMMSE-T 滤波器，这由是否要采用时空统计或时间统计决定。

假设噪声可以仿真为 $v(n_1,n_2,k)=s^a(n_1,n_2,k)\mu(n_1,n_2,k)$，其中 $\mu(n_1,n_2,k)$ 为零均值广义平稳过程，独立于信号，α 为一个实数。在这种模型下，令 $\mu_s(n_1,n_2,k)=\mu_g(n_1,n_2,k)$，且 $\sigma_s^2(n_1,n_2,k)=\sigma_g^2(n_1,n_2,k)-\sigma_v^2(n_1,n_2,k)$，其中 $\sigma_v^2(n_1,n_2,k)$ 表示噪声过程的时空方差。

实际上，整体均值 $\mu_g(n_1,n_2,k)$ 和方差 $\sigma_g^2(n_1,n_2,k)$ 用采样均值 $\hat{\mu}_g(n_1,n_2,k)$ 和方差 $\hat{\sigma}_g^2(n_1,n_2,k)$ 来代替，这两个值是在区域 $S_{n_1,n_2,k}$ 内通过计算而得到的（式（9-8）和式（9-9））。

$$\hat{\mu}_g(n_1,n_2,k)=\frac{1}{L}\sum_{(i_1,i_2;l)\in S_{n_1,n_2,k}} g(i_1,i_2;l) \tag{9-8}$$

$$\hat{\sigma}_g^2(n_1,n_2,k)=\frac{1}{L}\sum_{(i_1,i_2;l)\in S_{n_1,n_2,k}} [g(i_1,i_2;l)-\hat{\mu}_g(n_1,n_2,k)]^2 \tag{9-9}$$

式中，L 是 $S_{n_1,n_2,k}$ 中像素的个数。

$$\hat{\sigma}_s^2(n_1,n_2,k)=\max[0,\hat{\sigma}_g^2(n_1,n_2,k)-\hat{\sigma}_v^2(n_1,n_2,k)] \tag{9-10}$$

是为了避免可能有负方差估算值。

把这些估算值带入滤波器表达式（7-7）可以得到

$$\begin{aligned}\hat{s}(n_1,n_2,k)=&\frac{\hat{\sigma}_s^2(n_1,n_2,k)}{\hat{\sigma}_s^2(n_1,n_2,k)+\hat{\sigma}_v^2(n_1,n_2,k)}g(n_1,n_2,k)\\&+\frac{\hat{\sigma}_s^2(n_1,n_2,k)}{\hat{\sigma}_s^2(n_1,n_2,k)+\hat{\sigma}_v^2(n_1,n_2,k)}\hat{\mu}_g(n_1,n_2,k)\end{aligned} \tag{9-11}$$

式（9-11）体现了滤波器的自适应特性：当时空信号方差远小于噪声方差时，则 $\hat{\sigma}_g^2(n_1,n_2,k)\approx 0$，即区域 $S_{n_1,n_2,k}$ 是均匀的，$\hat{\mu}_g=\hat{\mu}_s$；当时空信号方差远大于噪声方差时，

$\hat{\sigma}_s^2(n_1,n_2,k) \approx \hat{\sigma}_v^2(n_1,n_2,k)$，由于运动估算不准或 $S_{n_1,n_2,k}$ 没有空间清晰边缘的存在，估值接近于噪声图像值以避免模糊。

该滤波器的一个弊端是滤波衰减，经常会在滤波图像上留下噪声点。为此又提出了“转换的 LMMSE 滤波器”，它通过在每个像素所选择的一组时间和时空区域之间转换，保持时空和时间 LMMSE 滤波的优点。如果 $S_{n_1,n_2,k}$ 上的方差小于噪声方差 $\hat{\sigma}_g^2(n_1,n_2,k)$，则用该区域进行滤波，否则选择 $\hat{\sigma}_g^2(n_1,n_2,k)$ 小于噪声方差的最大区域。比如自适应加权平均滤波器为了在 $S_{n_1,n_2,k}$ 中选择最一致的子集滤波，采用了隐含的结构。

2．自适应加权平均滤波器

自适应加权平均滤波器沿着运动轨迹在时空区域内计算图像值的加权平均，而权的大小是由优化一个标准函数来决定加权值，而且这个权值是根据运动估算的精度和运动轨迹区域的空间一致性而变化的。在对通过整个轨迹的足够精度的运动估算和空间一致性的情况下，时空滤波器区域内的图像值达到相等的权值，且 AWA 滤波器可以进行直接时空平均。当时空区域内的一个特写像素的值偏离于被多于一个临界值滤波的像素值时，它的权值减少，且把重点转移到在区域内保持图像值从而更好地与关键像素相一致。因此，AWA 滤波器尤其适合序列的有效滤波，其中包含具有变化的场景内容，如由于快速图像放大及电视摄像机画面改变。

AWA 滤波器可以定义为

$$\hat{s}(n_1,n_2,k) = \sum_{(i_1,i_2;l)\in S_{n_1,n_2,k}} \omega(i_1,i_2;l)g(i_1,i_2;l) \tag{9-12}$$

其中

$$\omega(i_1,i_2;l) \doteq \frac{K(n_1,n_2,k)}{1+a\max\{\varepsilon^2,[g(n_1,n_2,k)-g(i_1,i_2;l)]^2\}} \tag{9-13}$$

是区域 $S_{n_1,n_2,k}$ 内的权值，而 $K(n_1,n_2,k)$ 是一个标准常量，通过下式给出

$$K(n_1,n_2,k) = \left(\sum_{(i_1,i_2;l)\in S_{n_1,n_2,k}} \frac{1}{1+a\max\{\xi^2,[g(n_1,n_2,k)-g(i_1,i_2;l)]^2\}}\right)^{-1} \tag{9-14}$$

a（a>0）及 ξ 是滤波器的参数，这些参数按照以下规则来决定：

（1）像素强度差仅有噪声时，加权平均简化为直接平均，只需适当地选择参数 ξ^2。我们令 ξ^2 的值等于噪声的两倍。

（2）若如果对于特定的 $(i_1,i_2;l)\in S_{n_1,n_2,k}$ 而言，值 $g(n_1,n_2,k)$ 和 $g(i_1,i_2;l)$ 之间的方差大于 ξ^2，则 $g(i_1,i_2;l)$ 的分布被 $\omega(i_1,i_2;l) < \omega(n_1,n_2,k) = K/(1+\alpha\xi^2)$ 抑制。参数 α 是“罚”参数，它决定了方差 $[g(n_1,n_2,k)-g(i_1,i_2;l)]^2$ 的权的灵敏度。“罚”参数 α 通常被设为 1。

9.1.3 运动自适应滤波

在视频处理中，帧间噪声滤波可能提供一些优于帧内滤波的优点。帧间滤波器的一个特殊分类是时间滤波器，该滤波器在时间方向上采用一维滤波。原则上，时间滤波器可以避免空间及时间上的模糊。

1．直接滤波

直接滤波的最简单形式是帧平均，即把连贯帧中相同位置的像素取平均。直接时间平均非常适合于静态图像部分，既可消除噪声，又不损失空间图像分辨率。由此可见，在纯时间滤波中，为有效减少噪声可能需要大量帧，这就需要大量帧存储。时空滤波需要一个折中的考虑，既保证噪声滤除效果，又能有效地减少需要的帧存储数目。

虽然直接时间平均可以很好地应用于静态图像区域，但它也可能导致活动区域图像模糊和色度分离。如果滤波器运用了帧间运动信息，就可以避免这些损失。主要的问题是如何从因噪声而产生的时间变化中分辨出因运动而产生的时间变化。适合于运动的时空噪声滤波器的例子包括直接滤波器和顺序滤波器，其中有中值、加权中值和多级中值滤波器。

2．基于运动检测的滤波

在这种方法中，所选择的滤波器结构具有可按照运动检测信号来调整的参数。FIR 和 IIR 结构都可应用于运动自适应滤波。一个简单的运动自适应 FIR 滤波器例子可以由下式给出：

$$\hat{s}(n_1,n_2,k)=(1-\gamma)g(n_1,n_2,k)+\gamma g(n_1,n_2,k-1) \tag{9-15}$$

IIR 滤波器的简单例子有

$$\hat{s}(n_1,n_2,k)=(1-\gamma)g(n_1,n_2,k)+\gamma \hat{s}(n_1,n_2,k-1) \tag{9-16}$$

其中

$$\gamma \doteq \max\{0,\frac{1}{2}-a\,|\,g(n_1,n_2,k)-g(n_1,n_2,k-1)\,|\} \tag{9-17}$$

是运动检测信号，α 是换算常数。由此可见，当检测大运动时，这些滤波器倾向于终止滤波器以防止虚像的产生。FIR 结构限制了降噪能力，IIR 滤波器在这方面更为有效，但是 IIR 滤波器一般会引起傅里叶相位失真。

9.1.4 运动补偿上行变换

如果运动轨迹可以精确估算，则运动补偿滤波是标准上行变换的最佳方法。在整体恒速运动的情况下，运动补偿滤波器是线性不变的。上行变换理论通常有两种描述途径：其一为数字途径，将输入信号填零，然后将数字插入滤波器应用到填入信号中；其二为模拟途径，首先对模拟信号进行概念上的重构，然后对其重新采样。因为填零需要运动估算的量化，所以限制了运动轨迹的精确性。

运动补偿滤波的基本概念与“帧/场速率上行变换”和“去隔行”是相同的，它沿着通过遗漏像素的运动轨迹进行滤波。具体程序分为 3 步：运动估算、运动估算后处理、滤波器设计。

1．运动估算

图 9.5（a）和图 9.5（b）分别表示正向或反向块匹配，通常用来估算两帧间的运动矢量。在运动补偿上行变换中，需要估算通过遗漏像素单元的运动轨迹。若干运动估算被建议用于标准变换。其中图 9.5（c）描述了一种简单的“对称块匹配”方法。该方法中，两个已有的相邻帧/场 k 和 k–1 中的块被对称地移动，使得连接这两个块中心的线始终通过所关注的遗漏像素 (x_1, x_2)。

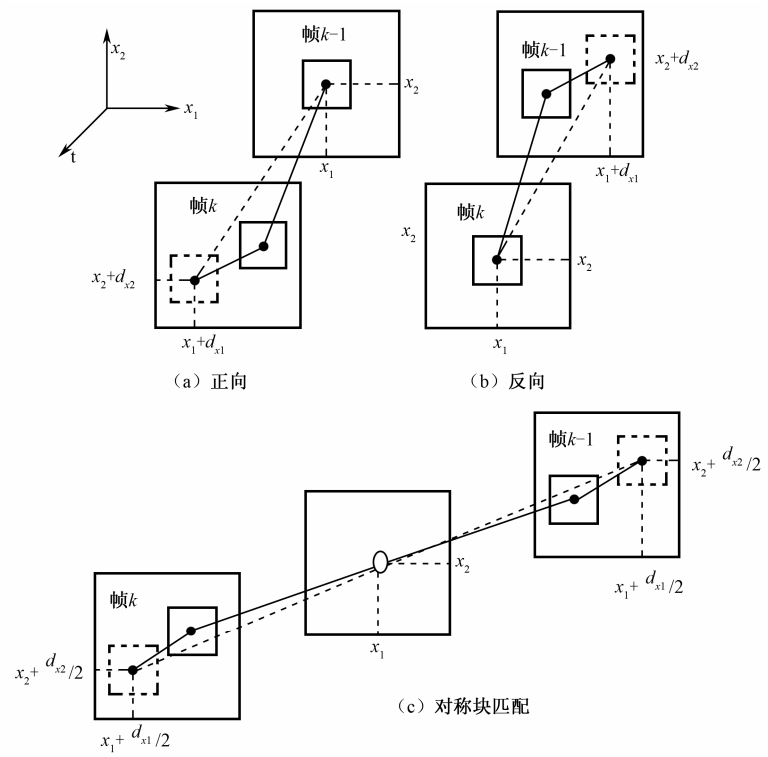


图 9.5　运动估算

运动估算精度也许是运动补偿插入效率中最重要的因素。因而常常将某类后处理应用于估算运动矢量以改善其精度。

2．运动估算后处理

通过后处理技术可以改善运动估算的精度。这里提出两种方法来测试运动估算的精度：遮断检测方法和位移帧差方法。必须指出的是，这些方法用于对称块匹配的两个非零帧。

遮断检测法是根据这样一个假设：通常运动矢量的冠（corona）均位于运动实体的周围。另外，观察从第 k 帧到第 k+1 帧的准确运动矢量总是映射到“变化区域”CD（k，k+1）。遮断检测步骤如下：如果从第 k 帧到第 k+1 帧的运动矢量估算映射到变化区域之外，则表示第 k 帧中一个像素将被第 k+1 帧覆盖；同样，如果从第 k+1 帧到第 k 帧的运动矢量映射到变化区域之外，就表示第 k+1 帧中一个像素未被覆盖。这种运动矢量当然是不可靠的，必须要丢掉。

对于多个满足基本条件的一组候选矢量，可以用候选运动矢量频率曲线的分析来确定：频率曲线的显著峰值表示了背景移动或在场景中大的物体移动。如果使用一个运动自适应或两个帧间滤波器，在一个像素上找不到可靠的转换运动矢量，则标志一个运动估算失败。

一般来讲，运动补偿滤波要求在每个像素上有不同的运动轨迹。但实际上，估计每个像素上的运动矢量不够可靠。对此影响，建议用混合去隔行方法，即针对因摄像机移动或抖动而引起的每一个单一整体运动进行补偿，然后对整体补偿的图像使用一种运动自适应滤波器，以计算任何残留运动。一种 3 个场混合去隔行滤波器的方框图如图 9.6 所示。其中假设 3 个连续场是：一个偶场 E_1、一个奇场 O_1、一个偶场 E_2。

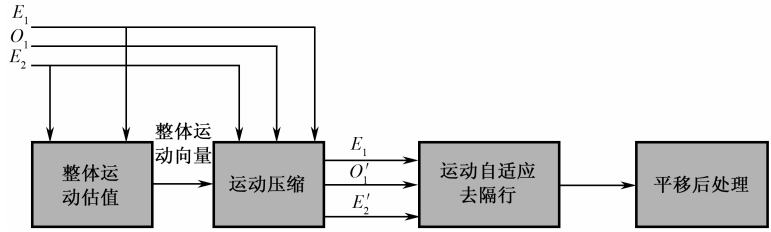


图 9.6　运动补偿/自适应去隔行

首先，在附近的 4 个矩形窗口上采用相位相关性方法来估算 E_1 和 E_2 场之间的整体运动矢量。其次，O_1 和 E_2 场是相对于 E_1 的运动补偿，并分别产生 O_1' 和 E_2'。运动补偿步骤的目的在于创建 3 个连续的场 E_1、O_1' 和 E_2'。如果不存在整体运动，则 3 个场表示隔行视频。然后将 3 场运动自适应加权平均滤波器应用到场序列 E_1、O_1' 和 E_2'。最后的后处理步骤中检测出由错误运动矢量引起的非线性虚像，相应的像素被空间插值所替换（如图 9.7 所示）。

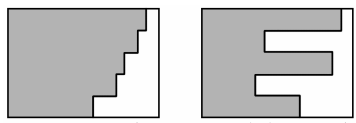


图 9.7　抖动图像不稳定示意图

此外，视频时空滤波还包括图像多帧复原、超分辨率等各种技术和应用，更多细节见参考文献[31,32]。

9.2　基于运动检测的自适应去交错

隔行到逐行扫描变换的基本方法就是利用已知的图像信息，通过线性内插方法重构出

每场缺少的行。一种获得插补行的最简单的方法是利用图像的空间相关性，直接利用场内上一行和下一行的信息平均来获得，即场内插补，如图 9.8 所示。该方法算法简单，运算量小，硬件要求相对较低，可以有效地消除闪烁，增加图像的细腻感。但它没有充分利用电视图像的时间相关性，图像的垂直清晰度并未提高。另一种方法是利用电视图像的时间相关性，将相邻两场的扫描线镶嵌，即场间插补。这种方法需要大容量的场存储器，可以消除闪烁，提高静止区域图像的垂直清晰度，但对运动图像会产生较严重的运动模糊。

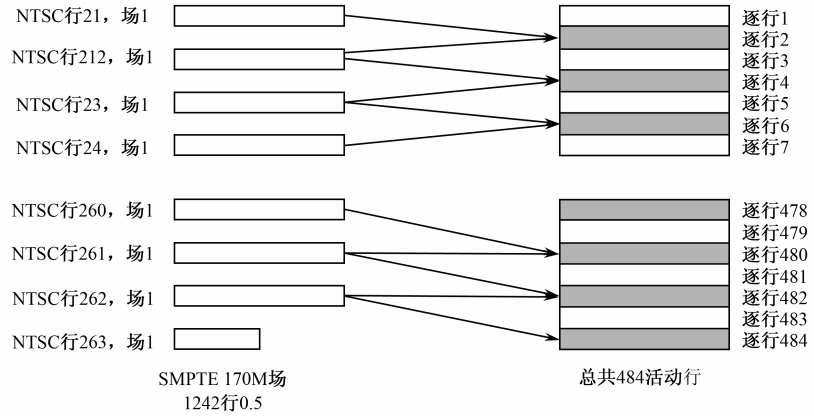


图 9.8　场内插补去交错技术示意图

这两种线性插补算法，对图像的所有部分采用同一算法。而实际图像既包括静止部分也包括运动部分，如果没有根据图像的局部特征而采用相应算法，则不能保证图像的所有部分均获得好的转换效果。因此，必须采用基于运动检测的自适应控制内插算法，即首先通过判断待插补点是静止还是运动，然后对静止图像采用场间插补，对运动图像采用场内插补，以达到最佳的转换效果。

为了进行帧间视频信号处理，需要同时保留相邻几帧的视频数据。由于数据量较大，一般存放在低速存储器（如 L3）中。同时为了提高处理速度，可将图像划分为多个子块，依次读入 L1 内存中进行处理。由于采用 DMA 方式，对处理器内核几乎不造成什么负担。在算法实现中，视频输入/输出缓冲区的定义、去交错处理输入/输出缓冲区的定义（在 L1 中）可以参考第 5 章的相关内容，这里不再详细介绍。此时我们需要同时读入至少两帧视频图像对应部分的内容，DMA 的处理上复杂度有所增加。

下面介绍基于运动检测的自适应控制内插算法的一个简单实现。它首先计算对应位置的差并统计其方差，然后判断每个位置上是否存在运动，最后对静止的位置进行帧内插值，对存在运动的位置采取帧间插值。函数实现分别如下。

（1）计算每个位置各个方向上的帧间差值，统计其方差。

```
void calc_diff( unsigned char * src1, unsigned char * src2,
          unsigned short* d0, unsigned short* d1, unsigned short* d2,
          unsigned short* d3, unsigned short* d4, unsigned short* var)
{
```

```
int index = (WIDTH<<2);                    // 宽度的 4 倍
int index2 = (WIDTH<<1);                   // 宽度的 2 倍
int i, j, ind, ind2, ind3, mean;
for( i = 4 ; i < HEIGHT-4 ; i += 2 ) { // 水平方向
    for( j = 2 ; j < WIDTH-2 ; j++ ) { // 垂直方向
        ind = index + j;                   // 图像像素索引
        ind3 = index2 + j;                 // 方差索引
        d0[ind3] = abs(src1[ind]-src2[ind]) + abs(src1[ind-1]-src2[ind-1] )
            + abs(src1[ind+1]-src2[ind+1])
            + abs(src1[ind-(WIDTH<<1)]-src2[ind-(WIDTH<<1)])
            + abs(src1[ind+(WIDTH<<1)]-src2[ind+(WIDTH<<1)])
            + abs(src1[ind-(WIDTH<<1)-1]-src2[ind-(WIDTH<<1)-1])
            + abs(src1[ind+(WIDTH<<1)+1]-src2[ind+(WIDTH<<1)+1])
            + abs(src1[ind-(WIDTH<<1)+1]-src2[ind-(WIDTH<<1)+1])
            + abs(src1[ind+(WIDTH<<1)-1]-src2[ind+(WIDTH<<1)-1]);
    ind2 = ind + 1;
    d1[ind3]= abs(src1[ind]-src2[ind2])+abs(src1[ind-1]-src2[ind2-1])
            +abs(src1[ind+1]-src2[ind2+1])
            +abs(src1[ind-(WIDTH<<1)]-src2[ind2-(WIDTH<<1)] )
            +abs(src1[ind+(WIDTH<<1)]-src2[ind2+(WIDTH<<1)])
            +abs(src1[ind-(WIDTH<<1)-1]-src2[ind2-(WIDTH<<1)-1])
            +abs(src1[ind+(WIDTH<<1)+1]-src2[ind2+(WIDTH<<1)+1])
            +abs(src1[ind-(WIDTH<<1)+1]-src2[ind2-(WIDTH<<1)+1])
            +abs(src1[ind+(WIDTH<<1)-1]-src2[ind2+(WIDTH<<1)-1]);
            ind2 = ind - (WIDTH<<1);
        d2[ ind3 ] = …                     // 同 d1
        ind2 = ind-1;
        d3[ ind3 ] = …                     // 同 d1
        ind2 = ind + (WIDTH<<1);
        d4[ ind3 ] = …                     // 同 d1
        mean =(src1[ind]+src1[ind+1]+src1[ind-1]+src1[ind+(WIDTH<<1)]
            + src1[ind+(WIDTH<<1)+1] + src1[ind+(WIDTH<<1)-1]
            + src1[ind-(WIDTH<<1)] + src1[ind-(WIDTH<<1)+1]
            + src1[ind-(WIDTH<<1)-1] ) / 9;
        var[ind3] = ((abs(src1[ind]-mean)
            +abs(mean-src1[ind+(WIDTH<<1)])
            +abs(mean-src1[ind-(WIDTH<<1)]) + abs( mean-src1[ind-1])
            +abs(mean-src1[ind+1])+abs(mean-src1[ind-(WIDTH<<1)-1])
            +abs(mean-src1[ind-(WIDTH<<1)+1])
```

```
                +abs(mean-src1[ind+(WIDTH<<1)-1] )
                +abs( mean-src1[ind+(WIDTH<<1)+1] ) ) >>3 );
        }
        index += (WIDTH<<1);
        index2 += WIDTH;
    }
}
```

（2）根据差值及方差判断对应位置是否存在运动。

```
void motion_judge (unsigned short* d0, const unsigned short* d1,
                   unsigned short* d2, unsigned short* d3,
                   unsigned short* d4, unsigned short* var)
{
 int i, j, index, ind;
 index = (WIDTH<<1);
 for(i=2; i<HALF_HEIGHT-2; i++) {
     for(j=1; j<WIDTH-1; j++) {
         ind = index+j;
         if( (d1[ind]+(var[ind]) < d0[ind]
             || (d2[ind]+var[ind]) < d0[ind]
             || (d3[ind]+var[ind]) < d0[ind]
             || (d4[ind]+var[ind]) < d0[ind] ) && var[ind] > 2 ) {
             num1++ ; motion_or_not[ind] = 1;    // 存在位移
         } else {
             num2 ++; motion_or_not[ind] = 0;    // 没发生位移
         }
     }
     index += WIDTH;
 }
}
```

（3）根据运动与否采取不同的插值处理。

```
void motion_adaptive_deinterlace(unsigned char * src1, unsigned char * src2)
{
 int i, j, ind;
 unsigned int index;
 index = (WIDTH<<2);
 ind=(WIDTH<<1);
 mean_interporate[0] = inter_mean;       // 帧间插值函数指针地址
 mean_interporate[1] = intra_mean;       // 帧内插值函数指针地址
```

```
    for( i = 4 ; i<HEIGHT-4 ; i+=2 ) {
        for(j=1; j<WIDTH-1; j++) {
            src1[index+j] = mean_interporate[ motion_or_not[ind+j] ]
                                                    ( src1, src2, index+j );
        }
        ind += WIDTH;
        index += (WIDTH<<1);
    }
}
```

帧间插值和帧内插值算法的具体实现可以灵活设计。

在上述算法中，为计算对应块之差的相似度，需要计算其绝对差之和（SAD），SAD 最小表示两个子块相似度最高，可由此计算出运动矢量。SAD 的计算是算法中的重要一环，它直接影响整个搜索过程的速度。Blackfin 专门提供了视频操作指令来完成该处理过程，利用该指令，只需要一条指令就可以取出 8 个单元数据，同时完成 4 字节的绝对差和计算。这条指令就是视频操作指令 SAA（R1:0，R3:2），处理结果存放到 A0 和 A1 中。这里 A0 和 A1 为两个累加器，R1 和 R0、R2 和 R3 为配对的两对寄存器（一般称为寄存器组 0 和寄存器组 1）。对于 16×16 宏块的 SAD 优化的部分汇编代码参考下述代码。由于绝对差和是针对字节而言，所以起始字节地址不一定是 4 的整数倍，但每次运算却必须要取出 4 字节作运算。为此，Blackfin 巧妙地根据地址的末两位来确定运算所取的字节。其地址指针为 I0 和 I1，由 I0 和 R1:0、I1 和 R3:2 来确定运算的 8 个点。

```
// 建立硬件循环，P2 寄存器存放循环次数 16
LSETUP (SAD_START, SAD_END) LC0=P2;
    // 读内存，一次取 32 位
    A1=A0=0 || R0 = [I0++] || R2 = [I1++];
SAD_START :
    // 寄存器 R0、R2 的 4 字节分别求绝对差值，并累加
    SAA (R1:0,R3:2) || R1 = [I0++]  || R3 = [I1++];
    // 寄存器 R1、R3 的 4 字节分别求绝对差值，并累加
    SAA (R1:0,R3:2) (R) || R0 = [I0++] || R2 = [I1++];
    SAA (R1:0,R3:2) || R1 = [I0 ++ M0] || R3 = [I1++M1];  //
SAD_END:
    SAA (R1:0,R3:2) (R) || R0 = [I0++] || R2 = [I1++];
    R3=A1.L+A1.H , R2=A0.L+A0.H ;
    R0 = R2 + R3 (S) ; // 得到累加和
```

由于可以在循环中不停地累加，而且在同一时钟周期可并行处理运算和更新地址内容，这样就大大节约了时间。实验表明，采用该指令可以使 SAD 的处理速度提高近 50 倍。

上述算法只是一种简单实现，可以在各个方面对其进行改进。比如可以扩大运动搜索范围，以适应活动程度更剧烈的视频信号。再比如可以实现 1/2 或 1/4 像素的亚像素运动矢

量，提高插值结果与真实值的逼近程度，进一步改进处理效果。上述处理方法可以明显提高去交错效果，但是也对 DSP 处理器性能提出了更高要求，因此更加适用于中高端的产品。针对具体产品的设计，则需要综合考虑成本、效果、电源等多种因素，做出最合适的选择。

9.3　视频滤波中的二维卷积运算

在视频时空滤波处理中要用到大量的各种二维图像卷积运算，本节讨论 Blackfin 处理器中二维卷积运算的实现[48]。

视频滤波应用中需要处理 8 位数据，因为单个像素分量（如 RGB 或 YUV）通常都是字节大小。因此，8 位视频 ALU 和基于字节的地址生成在处理像素时就具有了很大的差别。这一点不容小视，因为 DSP 一般是对 16 位或 32 位边界进行操作。另外，Blackfin 具有灵活的数据寄存器文件。在传统定点 DSP 中，字长通常是固定的。然而，数据寄存器既可以作为 32 位字（如 R0）使用，又能作为两个 16 位字（如 R0 的高半部分和低半部分 R0.H 和 R0.L），有很大的优点。

此外，专门的单周期指令对提供有效的多媒体编码算法非常方便。比如上面谈到的 SAA 指令。类似的操作都可以看做是二维图像卷积的某种具体实现。我们在此通过考查一个简单的基于二维卷积的图像滤波器组来说明二维卷积实现的基本方法和原则。

卷积是图像处理最基础的操作之一。在一个二维卷积中，对给定像素的计算是对其近邻像素亮度值的加权和。因为一个掩码的近邻是以给定像素为中心，掩码一般的尺寸是奇数的。掩码的尺寸相对于图像是很小的，如通常选择 3×3 掩码，因为这对逐像素处理在计算量上是适度的，但是已经大得足够检测图像的边缘。

3×3 内核的基本结构如图 9.9 所示。作为一个例子，对图像中第 20 行、第 10 列的像素的卷积处理的输出应该是：

```
Out(20,10) = A*(19,9) + B*(19,10) + C*(19,11) + D*(20,9) + E*(20,10)
           + F*(20,11) + G*(21,9) + H*(21,10) + I*(21,11)
```

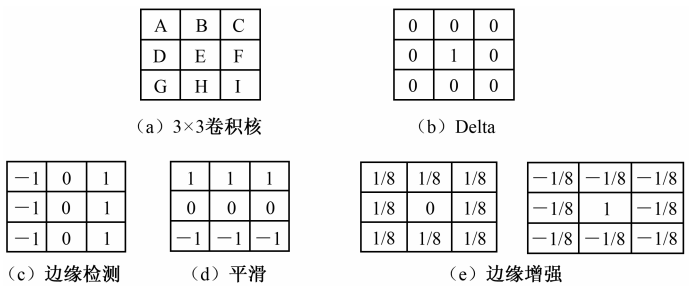


图 9.9　3×3 内核的基本结构

选择合适的系数能够有效的帮助计算。例如，如果尺度因子是 2 的乘方（包括分数），那么就可以用简单的移位操作来代替乘法操作。

图 9.9（b）～（e）显示了几个有用的 3×3 内核，每个都在下面有简单地介绍。图 9.9

（b）中的 Delta 函数是最简单的图像处理，将当前像素值输出而不加任何修改。图 9.9（c）显示了两个相邻行的边缘检测掩码。第一个掩码用于检测垂直边缘，第二个用于检测水平边缘。较高的输出值对应的位置边缘存在性高。图 9.9（d）中的内核是一个平滑滤波器，它对周围的 8 个像素取平均，将结果放到当前像素位置上。处理结果呈现为“平滑”的效果，或称为图像的低通滤波。图 9.9（e）中的滤波器称为锐化滤波器，它用当前位置的像素减去周围 8 个像素的平均值（即 d 中的平滑结果），能够产生一个边缘增强的图像。

仔细看一下二维卷积操作。高级别算法可以描述为如下步骤：

（1）将掩码的中心放到输入矩阵的一个元素上。

（2）将掩码邻居中的每个元素乘以对应的滤波器掩码元素。

（3）将每个成绩相加得到和结果。

（4）将得到的和放到输出矩阵中对应掩码中心的位置上去。

如图 9.10 所示为 3 个矩阵：输入矩阵 $f(x,y)$ 、一个 3×3 掩码矩阵 $h(x,y)$ 和一个输出矩阵 $g(x,y)$ 。

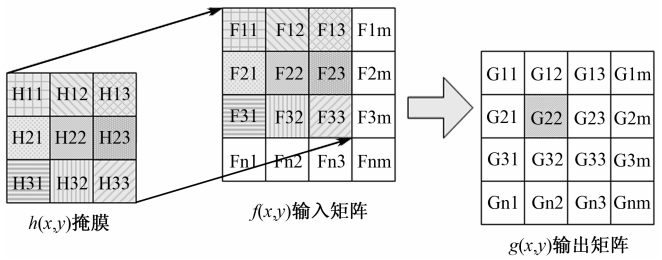


图 9.10　图像上的 3×3 卷积运算

计算完每个输出点后，将掩码向右移动一个元素。在图像边界上，将算法折回到下一行的第一个元素。例如，当掩码中心位于元素 F2m 时，掩码矩阵的元素 H23 要与输入矩阵的 F31 元素相乘。作为结果，输出矩阵的可用部分沿着图像的每条边要减 1。

图中对应的卷积公式是 $g(x,y)=\sum_{i=-1}^{1}\sum_{k=-1}^{1}h(i,k)f(x-i,y-k)$，对每个输出点（如 G22）的计算共需要 9 个乘法、8 个累加：H11*F11 + H12*F12 + H13*F13 + H21*F21 + H22* F22 + H23*F23 + H31*F31 + H32*F32 + H33*F33。

现在估算一下这个滤波器对处理器计算量的需要：对于一幅 VGA 图像（640×480 像素/帧），帧速率为 30 帧/s，每秒共有 9.2M 像素。如果这 9 个乘法和 8 个累加需要串行处理，即(9+8)×9.2=156MIPS；如果累加与乘法可以并行完成，则负载减少为 9×9.2=83MIPS。下面用一个例子显示如何进一步得到 2 倍的时钟周期节省。

对于上面所讲的代码，需要关注的是其内部循环，其中完成了所有的乘法/累加（MAC）操作。下面这个例子展示了通过正确地对齐输入数据，两个 MAC 单元可以在一个处理器周期内同时处理两个输出点。并且在同一个周期内，还可以在 MAC 操作的同时并行地完成多个数据的提取。该应用的关键部分是内部循环，如图 9.11 所示。

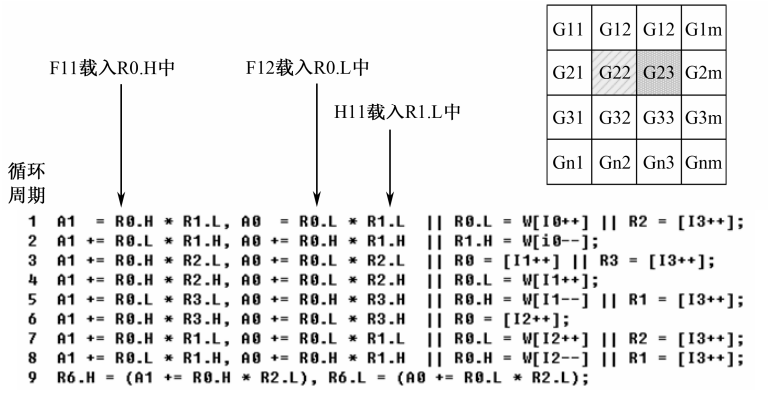


图 9.11　DSP 上 3×3 卷积的高效实现

对 G22、G23 的计算分别需要进行如下计算：G22 = H11*F11 + H12*F12 + H13*F13 + H21*F21 + H22* F22 + H23*F23 + H31*F31 + H32*F32 + H33*F33；G23 = H11*F12 + H12*F13 + H13*F14 + H21*F22 + H22* F23 + H23*F24 + H31*F32 + H32*F33 + H33*F34。利用图 9.11 中所示代码，则每循环得到两个输出点，9 个机器周期能够处理两个像素，所以平均每个像素需要 4.5 个周期。

内部循环的每一行都是一个单周期指令。输入数据表示为 16 位的数。输入矩阵的起始位置必须是 32 位边界对齐的。这就确保了输入矩阵中两个连续的点可以在单个 32 位读操作中完成。在进入该循环之前，输入矩阵的第一个值（F11）保存到 R0.H 中，第二个值（F12）保存到 R0.L 中，正如图 9.11 中周期 1 显示的前两个操作所示。寄存器 R1.L 也在进入内部循环之前被加载。它包含了掩码矩阵（H11）的第一个元素的值。

如前所述，为了得到输出矩阵的一个元素值，需要 9 个乘法和 8 个累加。然而，因为有双 MAC 操作，在每个内部循环中可以得到两个输出元素值。这样，F11×H11 和 F12×H11 在第一条指令结束时就保存到累计器中了。内部循环中的每条指令都移到下一个掩码值，结果保存在单独的累计器中，内部循环的最终输出载入到 R6 中。

每个机器周期不仅发生了多个算术运算，而且还并行地完成了多个数据载入/存储操作，以得到更高的效率。在此以机器周期 1 为例，下一个输入元素（F13）读入 R0.L，可以用在下一条指令的 MAC 操作中。类似地，R2 中载入下一组掩码值。这些值用在内部循环中接下来的 MAC 操作中。

视频滤波的其他相关技术和应用可以进一步参考相关资料[13~17]。

参 考 文 献

[1] 邓家先，康耀红. 信息量与编码. 西安：西安电子科技大学出版社，2007.

[2] 沈世镒，陈鲁生. 信息论与编码理论（第二版）. 北京：科学出版社，2010.

[3] 陈运. 信息论与编码（第 2 版）. 北京：电子工业出版社，2007.

[4] 方艳梅，刘永清译. 数字信号处理（第四版）. 北京：电子工业出版社，2007.

[5] 刘树棠，黄建国译. 离散时间信号处理（第 2 版）. 西安：西安交通大学出版社，2001.

[6] 张瑞峰，詹敏晶. 实用数字信号处理:从原理到应用. 北京：人民邮电出版社，2010.

[7] 贾洪峰译. 数字信号处理器——体系结构、实现与应用. 北京：清华大学出版社，2005.

[8] 阔永红译. 数字信号处理--基于计算机的方法. 北京：电子工业出版社，2006.

[9] 高梅国，刘国满，田黎育. 高速数字信号处理器结构与系统. 北京：清华大学出版社，2009.

[10] 朱志刚，林学阊，石定机等译. 数字图像处理. 北京：电子工业出版社，2010.

[11] 张大鹏. 模式识别与图像处理并行计算机系统设计. 哈尔滨：哈尔滨工业大学出版社，1998.

[12] 章毓晋. 图像工程上册——图像处理和分析. 北京：清华大学出版社，2001.

[13] 崔之祜，江春，陈丽鑫译. 数字视频处理. 北京：电子工业出版社，1998.

[14] 黎洪松. 数字视频处理. 北京：北京邮电大学出版社，2006.

[15] 谢剑斌，徐晖. 数字视频处理与显示. 北京：电子工业出版社，2010.

[16] 刘富强，王新红，宋春林. 数字视频图像处理与通信. 北京：机械工业出版社，2010.

[17] 李玉山. 数字视觉视频技术. 西安：西安电子科技大学出版社，2006.

[18] 姚春莲，周兵. 运动对象检测及其在视频压缩与处理中的应用. 北京：冶金工业出版社，2010.

[19] 高文，赵德斌，马思伟. 数字视频编码技术原理. 北京：科学出版社，2010.

[20] 刘峰. 视频图像编码技术及国际标准. 北京：北京邮电大学出版社，2005.

[21] Yao Wang，Jorn Ostermann，Ya-Qin Zhang. 视频信号处理与通信（英文影印版）. 北京：清华大学出版社，2003.

[22] 朱秀昌. 图像通信应用系统. 北京：北京邮电大学出版社，2003.

[23] 王相海，宋传鸣. 图像及视频可分级编码. 北京：科学出版社，2009.

[24] 欧阳合，韩军译. H.264 和 MPEG-4 视频压缩：新一代多媒体的视频编码技术. 长沙：国防科技大学出版社，2004.

[25] 毕厚杰. 新一代视频压缩编码标准--H.264/AVC. 北京：人民邮电出版社，2006.
[26] 精英科技. 视频压缩与音频编码技术. 北京：中国电力出版社，2001.
[27] 王华，张建等译. DSP 原理及其 C 编程开发技术. 北京：电子工业出版社，2005.
[28] 陈峰. Blackfin 系列 DSP 原理与系统设计（第 2 版）. 北京：电子工业出版社，2010.
[29] 陈峰. 基于 Blackfin DSP 的数字图像处理. 北京：电子工业出版社，2009.
[30] 曹小秋，赵焕军. ADI Blackfin 系列 DSP 处理器实验指导书. 北京：电子工业出版社，2008.
[31] 罗勇江，刘书明，肖科. VisualDSP++集成开发环境实用指南. 北京：电子工业出版社，2008.
[32] 冯小平等译. 基于微信号结构的嵌入式信号处理. 北京：电子工业出版社，2008.
[33] 刘书明. ADI DSP 应用技术集锦. 北京：电子工业出版社，2009.
[34] 苏涛，蔺丽华，卢光跃，张林让. DSP 实用技术. 西安：西安电子科技大学出版社，2002.
[35] 张雄伟，陈亮，徐光辉. DSP 芯片的原理与开发应用. 北京：电子工业出版社，2003.
[36] 王念旭. DSP 基础与应用系统设计. 北京：北京航空航天大学出版社，2001.
[37] Analog Devices, Inc. A Beginner's Guide to Digital Signal Processing, 2011. http://www.analog.com/en/processors-dsp/processors/beginners_guide_to_dsp/fca.html
[38] Analog Devices, Inc. Blackfin Processors : Data Sheets, 2011. http://www.analog.com/en/processors-dsp/blackfin/processors/data-sheets/resources/index.html.
[39] Analog Devices, Inc. Blackfin Processors : Manuals, 2011. http://www.analog.com/en/processors-dsp/blackfin/processors/manuals/resources/index.html.
[40] Analog Devices, Inc. Software Development Kit （SDK）, 2011. http://www.analog.com/en/processors-dsp/software-and-reference-designs/content/software_development_kit_downloads/fca.html.
[41] Analog Devices, Inc. Blakcfin Code Examples, 2011, http://www.analog.com/en/processors-dsp/Blackfin/processors/code-examples/Blackfin_Code_Examples/resources/fca.html.
[42] Analog Devices, Inc. Blackfin Image Processing Toolbox, 2011. http://www.analog.com/en/processors-dsp/blackfin/BF_IPTBX_00/processors/product.html.
[43] Analog Devices, Inc. Video Analytics Toolbox, 2011. http://www.analog.com/en/processors-dsp/blackfin/BF_OBJDT_00/processors/product.html.
[44] Analog Devices, Inc. H.264 Baseline Profile Encoder, 2011. http://www.analog.com/en/processors-dsp/blackfin/BF_H264_BP_ENCODER/processors/product.html.
[45] Analog Devices, Inc. H.264 Baseline Profile Decoder, 2011. http://www.analog.com/en/processors-dsp/blackfin/BF_H264_BP_DECODER/processors/product.html.

[46] Analog Devices, Inc. Video Framework Considerations for Image Processing on Blackfin Processors, 2011. http://www.analog.com/static/imported-files/application_notes/33507857201085EE276rev1_0905.pdf.

[47] Analog Devices, Inc. Video Templates for Developing Multimedia Applications on Blackfin Processors, 2011. http://www.analog.com/static/imported-files/application_notes/EE_301.pdf

[48] Analog Devices, Inc. Video Filtering Considerations for Media Processors, 2011. http://www.analog.com/static/imported-files/tech_articles/47593453337791VideoFiltering.pdf.

[49] Analog Devices, Inc. Blackfin Processor's Parallel Peripheral Interface Simplifies LCD Connection in Portable Multimedia, 2011. http://www.analog.com/static/imported-files/tech_articles/507257778149571cd_drive.pdf.

[50] Analog Devices, Inc. Interfacing Blackfin EZ-KIT Lite Boards to CMOS Image Sensors, 2011. http://www.analog.com/static/imported-files/application_notes/EE_300_Blackfin_Easy_KIT_Lite.pdf.

[51] 钟玉琢，沈洪，冼伟铨，田淑珍. 多媒体技术基础及应用. 北京：清华大学出版社，2006.

[52] 于仕琪，刘瑞祯译. 学习 OpenCV. 北京：清华大学出版社，2009.

[53] 王小鹏译. 形态学图像分析原理与应用（第 2 版）. 北京：清华大学出版社，2008.

[54] 崔屹. 图象处理与分析——数学形态学方法及应用. 北京：科学出版社，2002.

[55] Stan Z. Li, Anil K. Jain. Handbook of Face Recognition. New York: Springer. 2005.

[56] 于培松. 基于 BLACKFIN533 的 H.264/AVC 视频编码研究（硕士学位论文）. 西安：西安电子科技大学，2005.

[57] 艾孟奇. H.264 标准视频解码优化及 DSP 程序设计（硕士学位论文）. 成都：电子科技大学，2007.